HP64000
Logic Development
System

Pascal/64000 Reference Manual

HEWLETT
PACKARD

NOTICE

Attached to this software notice is a summary of problems and solutions for the Pascal Compiler Reference Manual that you may or may not encounter. Use this summary with the manual you received with the product. In the one-line description at the top of each problem and solution, there is a software topic or manual chapter reference.

KPR #: D200000638 Product: PASCAL REF M64800-90909 01.01

Keywords: SET

One-line description:
Negation of SET type will not produce error msg. (See Ch 2, pg 2-15)

Problem:
The statement A := INTEGER(-SETTYPE(X1)): will produce incorrect code and not produce an error. An error should result due to the attempt to negate a SET type designation.

Solution:
Be careful. PASCAL does not provide for the negation of a SET.

KPR #: D200001958 Product: PASCAL REF M64800-90909 01.01

Keywords: SETS

One-line description:
Error #407 for valid set exp with set const on left. (See Ch 4, pg 4-9)

Problem:
The compiler incorrectly performs the set construction for expressions.
Error 407 in Pass One of the compiler for expressions of the form

```
            [A] = B;
```

where B is a set of scalar type and A is an element of scalar type.
Error 407 also can appear for expressions such as:

```
         WHILE [SWO] >= [SWO,SW1] DO
```

Solution:
Rewrite expression putting the set construction on the right hand side of the expression as in

```
                      B = [A];
```

KPR #: D200014514 Product: PASCAL REF M64800-90909 01.01

Keywords: Y48

One-line description:
Module with many externals may cause system to reboot.(See Ch 1, pg 1-3)

Problem:
A module with many externals <>100) and an IF statement with 4 external arrays used may cause the system to reboot.

Solution:

Change the number of external references by adding one or two dummy references. Subtracting one or two unnecessary references can also cause the problem to disappear.

KPR #: D200033944 Product: PASCAL REF M64800-90909 01.01

Keywords: TYPE CONVERSION

One-line description:
Constants of type BYTE lose their "BYTENESS" when added.

Problem:

```
$EXTENSIONS;RANGE$
CONST C1 = BYTE(80H);   (*-128*)
      C2 = BYTE (1H);   (* + 1*)
VAR   B1 : BYTE;

BEGIN
B1 := C1 + C2;
{IF $RANGE$ IS ON, THIS CAUSES A CALL TO A WORD-SIZED, BOUNDS-CHECKING
ROUTINE PASSING 0081H AND THE VALUE IN QUESTION.  THIS PASSED VALUE IS
WRONG BECAUSE IT IS NOT SIGN-EXTENDED AND THE UPPER-BOUNDS IS 007FH.}
```

Solution:
1. $RANGE OFF$ around the code; or better yet use:

```
     B1 := BYTE(C1+C2);  {This calls the byte-sized bounds checker.}
```

KPR #: D200015842 Product: PASCAL REF M64800-90909 01.01

Keywords: MANUAL

One-line description:
I/O cannot be performed at absolute address 0000. (See Ch 6, pg 6-12)

Problem:
The 6800 supplement states that the PARAM_ routine cannot be passed a variable at absolute address zero (due to interpretation as an indirection flag), but the user must infer that the Pascal I/O routines use PARAM_ (it's not stated in the Pascal Manual.)

Solution:
Under subparagraph "Implementation Dependent Features" (page 6-12), add the following:

"Pascal I/O cannot be performed to variables at address 0000."

```
KPR #: D200035048   Product:    PASCAL REF          M64800-90909      01.01
```

Keywords: MANUAL

One-line description:
Better documentation concerning the use of UNSIGNED and SIGNED numbers.

Problem:
HP 64000 Pascal manuals need better descriptions of the use and constraints of UNSIGNED and SIGNED numbers. Included in these descriptions should be:

(1) When does the compiler "automatically" convert from one type to the other. When does the user have to force the type conversion.

(2) What are some of the SIGNED to UNSIGNED considerations with respect to the sign-bit of a number. What are some of the potential pitfalls when converting a SIGNED number into an UNSIGNED value.

(3) What are some potential pitfalls when converting REAL numbers into SIGNED or UNSIGNED values.

Solution:
In reference to the above questions, the following additional information should be entered into the manual on pages indicated.

Reference to question (1) above: Enter on page 2-9 in the manual, the following:

NOTE

The compiler never "automatically" converts from one type to the other. The user must force the type conversion.

Reference to questions (2) and (3) above, enter the following information on pages 2-9 and 2-10 in the manual:

When converting types, the following occurs:

```
from SIGNED_X
     to
a longer SIGNED_Y : the result is sign extended, value preserved.

  from SIGNED_X
        to
  a shorter SIGNED_Y : result is truncated; if original value does
                       not fit into smaller type, then value is
                       changed.
```

from UNSIGNED_X
to
longer UNSIGNED_Y : result is zero extended; value preserved.

from UNSIGNED_X
to
shorter UNSIGNED_Y : result is truncated.

from SIGNED_X
to
same size UNSIGNED_X : bits remain unchanged; interpretation may change value.

from UNSIGNED_X
to
same size SIGNED_X : bits remain unchanged; interpretation may change value.

from SIGNED_X
to
longer UNSIGNED_Y : result is sign extended; if original SIGNED_ was positive, then value is preserved; if original SIGNED_ was negative, then the resulting value is wrong.

from UNSIGNED_X
to
longer SIGNED_Y : result is zero extended; value preserved.

from any SIGNED_X
to
REAL or LONGREAL : result is sign conversion; however, it is possible to lose some significance.

from LONGREAL or REAL
to
any SIGNED_X : possible that original value will not fit into receiving type resulting in truncation.

from UNSIGNED_ types (except longest)
to
REAL or LONGREAL : sign conversion.

from longest UNSIGNED_
to
REAL or LONGREAL: sign conversion; wrong answer if most signif- icant bit is set.

from REAL or LONGREAL
to
any UNSIGNED_ type : sign conversion; possible value will not fit into receiving type - truncated result; if original value negative, then result is wrong.

KPR #: 2700005132 Product: PASCAL REF M64800-90909 01.01

Keywords: PASS 1

One-line description:
BYTE to UNSIGNED_16 assignments may not work. (See Chap 3, pg 3-16)

Problem:
Additional information needed when making BYTE to UNSIGNED_16 assignments.

Solution:
The following information modifies paragraph 3 on page 4-4 in the Pascal reference manual:

> When converting a signed operand to a physically larger signed or unsigned operand, the smaller signed operand will be sign extended by the compiler. When converting an unsigned operand to a physically larger signed or unsigned operand, the smaller unsigned operand is extended with zeroes.

The following information should be added to page 2-5 in the Pascal reference manual:

> All signed and unsigned integer CONSTant declarations are 32 bits long. The type of the constant is derived from its value. Type changes set basic types (i.e. signed, unsigned, CHARacter, set, enumerated type); however, the size of the type is derived from the value. Several examples:

```
        A = 85;              (* SIGNED_8 *)
        A = BYTE(80H);       (* SIGNED_16 value is 128 *)
        A = BYTE(300);       (* SIGNED_16 *)
        A = UNSIGNED_8(255); (* UNSIGNED_8 *)
```

KPR #: D200007161   Product: PASCAL REF   M64800-90909   01.01

Keywords: PASS 1

One-line description:
COMP_SYM file not automatically purged. (See Chap 9, pg 9-4)

Problem:
If COMP_SYM option is not selected when compiling, previously created COMP_SYM file is not purged. This file is used by the 6433x high level software analyzers. Since the COMP_SYM file is not automatically purged it is possible that an old COM_SYM file that does not apply to the code being executed and analyzed might be used giving incorrect results during analysis.

Solution:
Purge the COMP_SYM file before compiling.

KPR #: D200045583   Product: PASCAL REF   M64800-90909   01.01

Keywords: MANUAL

One-line description:
$LIST_OBJ$ inop without $LIST_CODE$ or expand opt. (See Ch 8, pg 8-5)

Problem:
"6800"

```
{  PascBug27:  Directive LIST_OBJ has no effect unless LIST_CODE
               specified.

   The Pascal manual states that the LIST_OBJ directive " ... causes
   the listing to contain the relocatable object code generated by the
   third pass of the compiler."  However, specifying LIST_OBJ will not
   produce the object code unless LIST_CODE or "options expand" is
   also specified.

   To observe the problem, try:
     "compile PascBug27 listfile display"    }

 PROGRAM PascBug27;
 $ LIST_OBJ ON $
 BEGIN
 END.
```

Solution:
In manual, use the following description for $LIST_OBJ$:

```
     LIST_OBJ   [ON]
                [OFF]
```

Initialized Value: OFF

Description:

ON causes the listing to contain the relocatable object code generated by the third pass of the compiler. In order for this directive to be effective, the LIST_CODE directive or the compile option 'expand' must be ON. This directive is not implemented in the 8080/8085 and the Z80 compilers.

KPR #: D200047225 Product: PASCAL REF M64800-90909 01.01

Keywords: MANUAL

One-line description:
$EMIT CODE OFF$ has no effect. (See Chap 8, pg 8-3)

Problem:
The Pascal manual states that the EMIT_CODE directive can be used to specify whether or not executable code is emitted to the relocatable file. However, EMIT_CODE directives are ignored, and the code is always produced.

The alternative method of suspending code emission, that of commenting out the area, may not be possible if the code contains comments, as Pascal comment indicators do not nest.

Solution:
Add the following sentence to the end of the description of the compiler option 'EMIT_CODE':

> "The last value of this option in a source file determines if code is generated for the entire file."

FOLD HERE

HEWLETT
PACKARD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 1303 COLORADO SPRINGS, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

HEWLETT-PACKARD
Logic Product Support Dept.
Attn: Technical Publications Manager
Centennial Annex - D2
P.O. Box 617
Colorado Springs, Colorado 80901-0617

FOLD HERE

Your cooperation in completing and returning this form will be greatly appreciated. Thank you.

READER COMMENT SHEET

Operating Manual
Pacal/64000 Reference Manual
64800-90909, January 1984

Your comments are important to us. Please answer this questionaire and return it to us. Circle the number that best describes your answer in questions 1 through 7. Thank you.

1. The information in this book is complete:

Doesn't cover enough (what more do you need?)	1	2	3	4	5	Covers everything

2. The information in this book is accurate:

Too many errors	1	2	3	4	5	Exactly right

3. The information in this book is easy to find:

I can't find things I need	1	2	3	4	5	I can find info quickly

4. The Index and Table of Contents are useful:

Helpful	1	2	3	4	5	Missing or inadequate

5. What about the "how-to" procedures and examples:

No help	1	2	3	4	5	Very helpful
Too many now	1	2	3	4	5	I'd like more

6. What about the writing style:

Confusing	1	2	3	4	5	Clear

7. What about organization of the book:

Poor order	1	2	3	4	5	Good order

8. What about the size of the book:

too big/small	1	2	3	4	5	Right size

Comments: ______________________________

____

____

____

Particular pages with errors?

____

Name (optional): ______________________________
Job title: ______________________________
Company: ______________________________
Address: ______________________________

Note: If mailed outside U.S.A., place card in envelope. Use address shown on other side of this card.

FOLD HERE

HEWLETT
PACKARD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 1303 COLORADO SPRINGS, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

HEWLETT-PACKARD
Logic Product Support Dept.
Attn: Technical Publications Manager
Centennial Annex - D2
P.O. Box 617
Colorado Springs, Colorado 80901-0617

FOLD HERE

Your cooperation in completing and returning this form will be greatly appreciated. Thank you.

READER COMMENT SHEET

Operating Manual
Pascal/64000 Reference Manual
64800-90909, January 1984

Your comments are important to us. Please answer this questionaire and return it to us. Circle the number that best describes your answer in questions 1 through 7. Thank you.

1. The information in this book is complete:

Doesn't cover enough (what more do you need?)	1 2 3 4 5	Covers everything

2. The information in this book is accurate:

Too many errors	1 2 3 4 5	Exactly right

3. The information in this book is easy to find:

I can't find things I need	1 2 3 4 5	I can find info quickly

4. The Index and Table of Contents are useful:

Helpful	1 2 3 4 5	Missing or inadequate

5. What about the "how-to" procedures and examples:

No help	1 2 3 4 5	Very helpful
Too many now	1 2 3 4 5	I'd like more

6. What about the writing style:

Confusing	1 2 3 4 5	Clear

7. What about organization of the book:

Poor order	1 2 3 4 5	Good order

8. What about the size of the book:

too big/small	1 2 3 4 5	Right size

Comments: ______________________________

____

____

____

Particular pages with errors?

____

Name (optional): ______________________________
Job title: ______________________________
Company: ______________________________
Address: ______________________________

Note: If mailed outside U.S.A., place card in envelope. Use address shown on other side of this card.

HEWLETT PACKARD

OPERATING MANUAL

PASCAL/64000 REFERENCE MANUAL

Manual Part No. 64800-90909

PRINTED: JANUARY 1984
CHANGED: OCTOBER 1985

Printing History

Each new edition of this manual incorporates all material updated since the previous edition. Manual change sheets are issued between editions, allowing you to correct or insert information in the current edition.

The part number changes only when each new edition is published. Minor corrections or additions may be made as the manual is reprinted between editions. Vertical bars in a page margin indicates the location of reprint corrections.

First Printing .... July 1980 (Part Number 64800-90904)
Reprinted .... November 1980
Second Edition .... May 1981 (Part Number 64800-90906)
Third Edition .... June 1982 (Part Number 64800-90908)
Fourth Edition .... January 1984 (Part Number 64800-90909)
Change U1085 .... October 1985

Table of Contents

Table of Contents (Cont'd)

Table of Contents (Cont'd)

Table of Contents (Cont'd)

Table of Contents (Cont'd)

Table of Contents (Cont'd)

Table of Contents (Cont'd)

List of Tables

List of Illustrations

List of Illustrations (Cont'd)

NOTES

Chapter 1

GENERAL INFORMATION

INTRODUCTION

This manual provides a description of the Pascal/64000 compiler and its operation on the HP Model 64000 Logic Development System. A description of the compiler options and their use is also included. Microprocessor dependent features of the compiler are documented in processor-dependent supplements.

NOTE

Refer to the Installation and Configuration Reference Manual for BOOT-UP operations and SOFTWARE UPDATING PROCEDURES when updating the system from a flexible disc.

PASCAL/64000 COMPILER

The Pascal/64000 compiler is an application program that translates Pascal/64000 source programs into relocatable object files, and, optionally, generates a listing file.

Pascal/64000 is an Implementation of a subset of the Pascal programming language "standard", defined by Jensen and Wirth in PASCAL User Manual and Report (second edition) published by Springer & Verlag, 1976. The language has been enhanced to improve its utility as a tool for microprocessor system programming.

The Pascal/64000 compiler uses a three-pass compilation process to translate source programs directly into relocatable code for the target microprocessor. Relocatable files for a particular microprocessor may be linked together to produce an absolute program file. Then, by using the emulator, the absolute file can be loaded into emulation memory and executed in the proper microprocessor environment.

An additional special preprocessor pass is available which gives the user the ability to have INCLUDE files, conditional compilation, and macros.

SUBSET LIMITATIONS

Since Pascal/64000 is a subset of Jensen and Wirth "standard" Pascal, there are some limitations. Pascal/64000 is compatible with the "standard" except for the subset limitations listed in the following subparagraphs:

- Strings in Pascal/64000 are defined differently than in standard Pascal. A Pascal/64000 string has a maximum length of 255 characters. In addition, it's length is dynamic (as opposed to fixed in standard Pascal) and can change during program execution.

- Packing of data is not carried below the byte level; the key word, **PACKED**, is ignored by the compiler, except when defining a string, and the standard procedures **PACK** and **UNPACK** are not implemented. Bit packing may be achieved in 8-, 16-, or 32-bit data items by using the **SET** construct "*" (AND), "+" (OR), and "-" (set difference), predefined **SHIFT** function, and functional type change operations.

- Sets are limited to 256 elements or less of any ordinal type. Set expressions with integer elements will be interpreted as being members of the SET OF 0..255 unless specifically qualified. Subsets are allowed, but the value of the maximum element of the **SET** may not exceed 255 (i.e., SET OF 240..255).

- Procedures and functions are not allowed as parameters at this time.

- Integers are 16-bit signed numbers in the range -32768..32767. 32-bit signed integers are implemented for the selected microprocessors. Refer to the processor-dependent manual supplements for specific information.

- Set subranges are not implemented.

- The standard function **SQR** is not implemented.

PASCAL/64000 EXTENSIONS

Pascal/64000 contains enhancements that provide more versatility for microprocessor programming. Since these enhancements are not part of standard Pascal, they are explained in the following paragraphs.

The CASE Statement

The **CASE** statement may contain an optional **OTHERWISE** clause to specify action to be taken if the value of the **CASE** expression is not equal to any of the constant values in the **CASE** statement.

CONST, TYPE, and VAR Declarations

The **LABEL** declaration, if used, must be placed ahead of the other declarations in the block of the program. The **CONST, TYPE,** and **VAR** declaration sections may follow in any order and may be repeated as often as required. The repetition of **CONST, TYPE,** and **VAR** declarations (in any order) is a Pascal/64000 extension to the standard Pascal.

Identifiers

In standard Pascal, only the first eight characters of two identifiers are compared to see if they are the same. In Pascal/ 64000, the first 15 characters are compared.

Letter cases in Pascal/64000 are significant while they are ignored in standard Pascal. In Pascal/64000, the identifier TEMP and Temp are different.

NEW and DISPOSE Procedures

In standard Pascal, procedures **NEW** and **DISPOSE** are used to control the area of memory required for dynamic storage. This area of memory is commonly referred to as the heap. It is sometimes extremely difficult to keep track of all the dynamic variables in a program merely to dispose of them when they are no longer required. Pascal/64000 provides two additional procedures, **MARK** and **RELEASE**, which simplify heap management in many situations. When it is known in advance that a group of dynamic variables may be needed on a short-term basis, the state of the heap, before the short-term variables are allocated, can be recorded by using the procedure **MARK. MARK(p)** is a predefined procedure having one parameter, a pointer variable, that records the heap state at the time MARK is executed. Calling **MARK(p)** causes assignment of the first free address in the heap to (*p*). Any execution of the procedure **NEW** will build new data structures, starting at the address held in (*p*).

When the short-term variables are no longer required, the heap can be returned to its original condition by using the procedure **RELEASE. RELEASE** is a predefined procedure having one parameter, a pointer variable, that restores the heap to the state present at the time of **MARK(p)**. All dynamic variables created after **MARK** was called, are effectively destroyed and the memory space occupied by those variables is available for allocation to new dynamic variables. The procedure **RELEASE** may return a large section of memory containing many dynamic variables. Therefore, it may have the effect of many calls to **DISPOSE**.

External and Global Declarations

Pascal/64000 allows separate compilation of modules which permit procedures and variables to be declared *EXTERNAL* or *GLOBAL* in a Pascal module so they may be defined or accessed in other Pascal, C, or assembly language modules.

Separate Relocatable Area Assignments

Pascal/64000 allows program code and constants to be compiled to a separate relocatable area from data and variables, permitting the design of ROM and RAM memory systems.

Absolute Memory Assignment

In addition to external linking, variables may be assigned to absolute memory locations permitting easy access to memory-mapped I/O addresses.

Constant-value Expressions

Wherever standard Pascal allows a constant, Pascal/64000 allows a constant expression. A constant expression is an expression containing operators and standard functions, constants which have already been defined, and literals.

Functional Type Conversion

Expressions or variables are permitted to have their implicit type changed by being included as the parameters of a "function" call of any named **TYPE**. Standard Pascal allows this by the use of variant type records, but the explicit functional type change greatly improves the readability of the source program.

Example:

```
VAR

  avgmph, time : INTEGER; travel: CHAR;
```

then when the statement:

```
avgmph := time * INTEGER(travel);
```

is executed, the CHAR variable, travel, is converted to INTEGER and multiplied by the INTEGER variable, time.

ADDR Function

The built-in function **ADDR** returns a value compatible with any pointer type. Standard Pascal never allows the user access to address information.

With the use of the **ADDR** function, pointers may point to any variable. Standard Pascal only allows pointers to dynamically allocate memory space.

SHIFT and ROTATE Functions

The built-in functions, **SHIFT** and **ROTATE**, provide for logical and circular shifting of data.

Predefined Data Type

The predefined data types BYTE, (or SIGNED__8), SIGNED__16, and SIGNED__32 represent 8-bit, 16-bit, and 32-bit signed integers respectively.

The predefined data types UNSIGNED__8, UNSIGNED__16, and UNSIGNED__32 represent 8-bit, 16-bit, and 32-bit unsigned integers respectively.

$USER__DEFINED$ Option

With the **$USER__DEFINED$** option, the user may selectively redefine the meaning to the arithmetic operators (+, -, *, /, DIV, =, <>, <, >, <=, and >=). For example, (*) may be redefined to do matrix multiplication when its operands are two-dimensional matrices.

PASCAL/64000 ENVIRONMENT

The Pascal/64000 compiler will run on any HP 64000 system that has memory expansion. The compiled code may be run using the proper emulation subsystem for the target microprocessor or on any independent system which uses the same target microprocessor. The following paragraphs list the Pascal/64000 character set and data types.

CHARACTER SET

Alphabetic characters	- All upper and lower case characters (A through Z and a through z).
Numeric characters	- Digits 0 through 9 for decimal numbers, including A through F for hexadecimal numbers.
Special characters	- Blank, dollar sign, apostrophe, left and right parentheses, comma, plus, minus, equals, less than, greater than, decimal point, slash, colon, semi-colon, left and right brackets, left and right braces, caret, asterisk, and underscore (__).

DATA TYPES

Intrinsic Scalar Data Types

BOOLEAN	- An 8-bit value representing the value **TRUE** (1) or **FALSE** (0).
BYTE	- An 8-bit signed integer in the range -128 to +127.
CHAR	- An 8-bit value in the set of characters defined by the 8-bit ASCII character set.
INTEGER	- A 16-bit or 32-bit signed number (depending on the micro-processor used).
LONGREAL	- A 64-bit binary value representing a floating-point number in IEEE double precision format.
REAL	- A 32-bit binary value representing a floating-point number in IEEE single precision format.
SIGNED__8	- An 8-bit signed interger in the range -128 to +127.
SIGNED__16	- A 16-bit signed integer in the range -32768 to +32767.

Intrinsic Scalar Data Types (Cont'd)

SIGNED__32	- A 32-bit signed integer in the range from -2,147,483,648 to +2,147,483,647. (Not defined for the 6800 and 6809 microprocessors.)
STRING	- A 256-byte array equivalent to PACKED ARRAY [0..255] of CHAR. Byte 0 contains the run-time string length.
UNSIGNED__8	- An 8-bit unsigned integer in the range 0 to 255.
UNSIGNED__16	- A 16-bit unsigned integer in the range 0 to 65535.
UNSIGNED__32	- A 32-bit unsigned integer in the range 0 to 4,294,967,295. (Not defined for the 6800 and 6809 microprocessors.)

User-definable Data Types

ARRAY TYPE	- A structure consisting of a fixed number of components which are all of the same type (called the component type), in which the components (elements of the array) are accessed by index expressions. The array type definition specifies the component type and the index type. The component type may be of any type, allowing "ARRAY OF ARRAY (OF ARRAY..)" to represent multi-dimensioned arrays with an arbitrary number of indices. The index type must be a simple type such as a scalar or subrange type.
POINTER TYPE	- A type representing a storage address for the target microprocessor.
RECORD TYPE	- A structure consisting of a fixed number of components, called fields, each of which can be of any type. For each field, the record definition specifies a field name identifier and the field type.
SCALAR TYPE	- A type that defines an ordered set of values by enumerating the identifiers which denote these values.
SET TYPE	- A structure defining the set of values that is the power set of its base type, (i.e., the set of all subsets of values of the base type). The base type must be a scalar or subrange type.

User-definable Data Types (Cont'd)

STRING TYPE	- A special class of arrays defined as PACKED ARRAY [0..n] of CHAR (where n <= 255).
SUBRANGE TYPE	- A type that is identified as a subrange of a previously defined ordinal type (char, byte, integer, or scalar) in which the smallest and largest values are user defined.
TEXT TYPE	- The type text is provided for doing common types of character- and line-oriented input and output. Variables of type TEXT are termed textfiles. Each component of a textfile is of the type CHAR, but the sequence of characters in a textfile is divided into lines. All operations applicable to a FILE OF CHAR can be performed on textfiles.

NOTE

Keywords must be all upper-case letters. Predefined identifiers such as **INTEGER, BYTE, ADDR**, and **SHIFT** must be all uppercase letters. User defined identifiers may be any combination of upper- or lower-case letters. Only the first 15 characters of an identifier are significant.

MANUAL ORGANIZATION

This manual fully documents the specific Pascal/64000 implementation of Jensen and Wirth Pascal. The reader wishing to learn Pascal should refer to an introductory text.

Chapters 2 thru 4 of this manual discuss the features of Pascal/64000 in a top-down fashion, starting with programs and ending with expressions.

Chapter 5 discusses the Pascal/64000 programming features that must be considered when writing source programs.

Chapter 6 discusses Input/Output characteristics.

Chapter 7 presents standard procedures and functions supported by Pascal/64000.

Chapter 8 discusses the Pascal/64000 compiler options.

Chapter 9 explains how to compile a Pascal/64000 program.

Chapter 10 explains how to link program modules compiled with the Pascal/64000.

Appendix A lists the Pascal/64000 compile-time and run-time errors.

NOTES

Chapter 2

GENERAL FORM OF A PASCAL PROGRAM

INTRODUCTION

Every Pascal/64000 program consists of a main program module and may contain as many procedure and function routines as necessary to properly execute the program. A program contains a heading, a declaration section, an optional compound statement, and concludes with a period (.) as indicated in figure 2-1.

Pascal/64000 allows a number of Pascal programs and assembly language modules to be linked together to form an executable program. If the compound statement is included in the program, an entry point is defined for the executable program. Execution begins at the beginning of this compound statement. If the compound statement is omitted, no entry point is defined. In this instance, the program is a collection of variables and routines that may be called and used by other programs. The omission of the program compound statement is an HP extension of Standard Pascal.

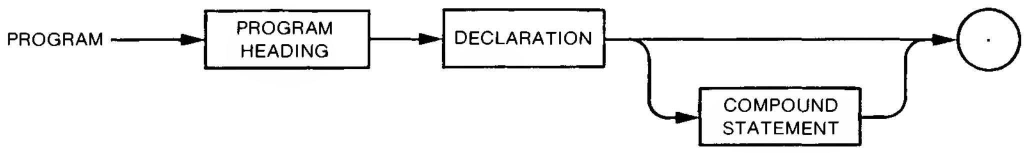


Figure 2-1. Program Syntax

PROGRAM HEADING

The word **PROGRAM** is a reserved word and is always the first word of a Pascal/64000 program heading. The program identifier (program name) follows the reserved word **PROGRAM** as shown in figure 2-2.

Pascal/64000 does not allow a program parameter list. The existence of program parameters implies the existence of an operating system which calls or executes the Pascal program. Such an operating system may not exist in a microprocessor environment.

Example:

```
PROGRAM NEWRS232;
   .
   .
   .
```

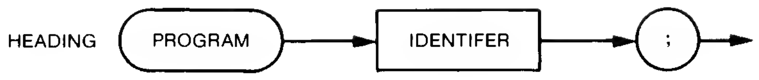


Figure 2-2. Heading Syntax

PROGRAM BLOCK

The program block contains declarations and statements (optional). The declaration section consists of definitions of labels, constants and types, declarations of variables, procedures and functions. Identifiers and labels declared in the main program block are known as global identifiers and labels. The statement part is made up of a compound statement which may be empty or may contain several simple or structured statements.

PASCAL PROGRAM DECLARATIONS

General Information

Every program consists of a heading and a block. The heading has been discussed previously. The block consists of a declaration section, and a statement section (optional) that specifies the action to be executed. Items identified in the program declaration are considered to be global in scope.

A complete block contains the following parts:

a. <LABEL declaration>
b. <CONSTant declarations>
c. <TYPE definitions>
d. <VARiable declarations>
e. <PROCEDURE and FUNCTION declarations>
f. <statements> (optional)

The declaration syntax is shown in figure 2-3.

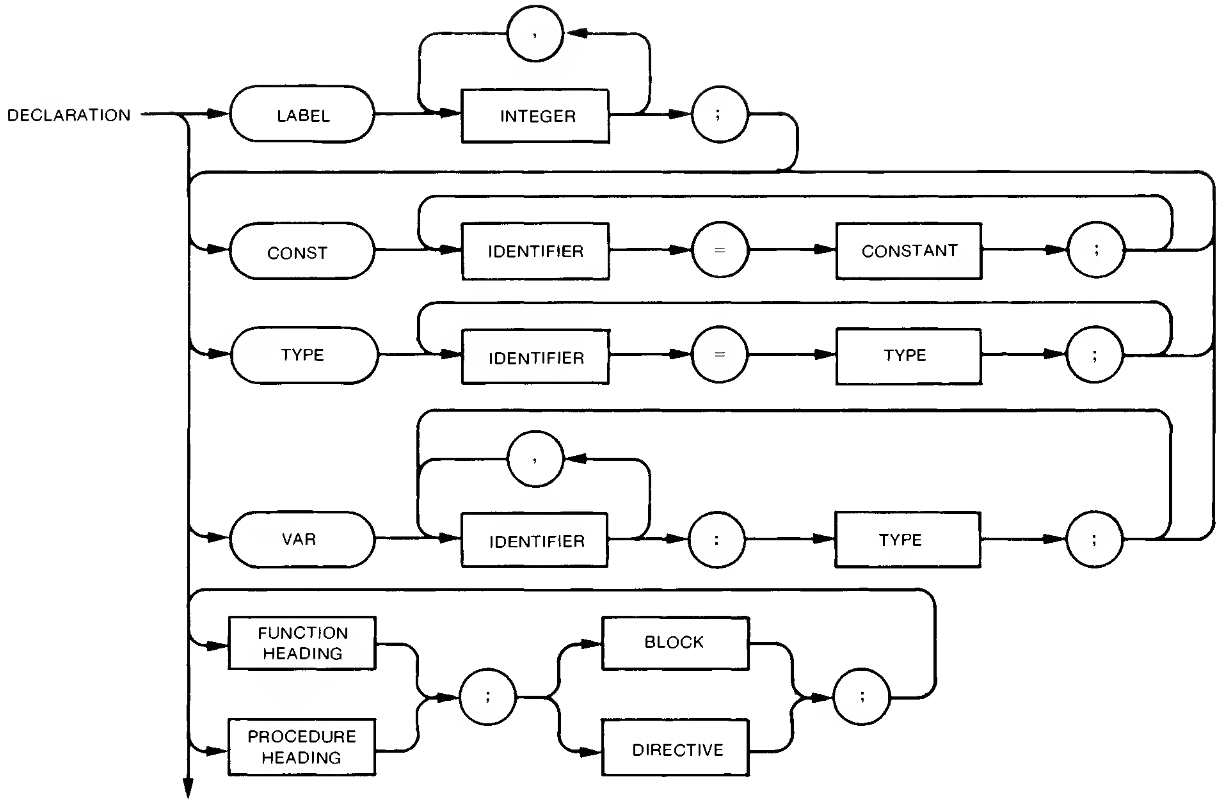


Figure 2-3. Declaration Syntax

The **LABEL** declaration, if used, must precede the other declarations in the block. The **CONST**ant, **TYPE**, and **VAR**iable parts may follow in any order and may be repeated as often as required. The repetition of **CONST**, **TYPE**, and **VAR** declarations is an HP extension to standard Pascal. The **PROCEDURE** and **FUNCTION** parts then follow in any order and may also be repeated as often as required.

Each part of the declaration section is discussed in detail in the following paragraphs. The statement section is discussed in Chapter 3.

DECLARATION SECTION

Label Declaration

Any statement in a program body may be identified by a label. Prior to use, however, the label must be identified in the **LABEL** declaration section of the block. When used, the **LABEL** declaration section must be placed before the other declaration sections in the block. Figure 2-4 shows the **LABEL** declaration syntax.

A label is an unsigned integer, no more than four digits long (1 through 9999) where leading zeros are not significant. Every label declared in the **LABEL** declaration section must subsequently be defined in the body of the block where the label is declared.

The function of the label is to identify a statement. Once identified, program control may be transferred to that statement by using a **GOTO <label>** statement. The following example illustrates both the **LABEL** declaration and label use.

```
PROGRAM showlabel;

LABEL
   1234;
VAR
   a,b:integer;
BEGIN
       .
       .
IF a > b
   THEN
      GOTO 1234
   ELSE
       .
       .
       .
1234: WRITELINE
END. {showlabel}
```

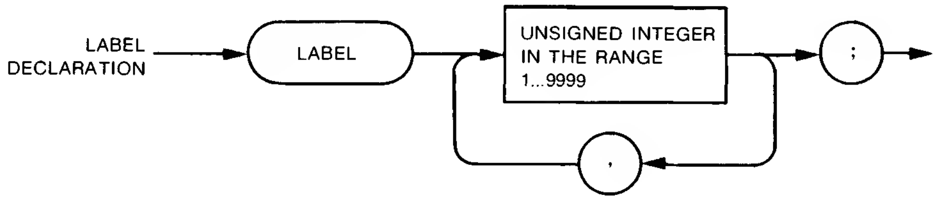


Figure 2-4. LABEL Declaration Syntax

CONSTant Declaration

The reserved word **CONST** precedes one or more constant definitions. A constant definition consists of an identifier, the equals sign (=), and a constant value.

A **CONST**ant definition introduces an identifier as a synonym for a constant value. A constant may denote the value for an ordinal type, a real type, a pointer type, a set type, or a string type. **CONST**ants may not denote values for an array type, a record type, or a file type.

NOTE

Constants are assumed to be signed entities. Their magnitude determines whether they are considered to be signed_8, signed_16, or signed_32.

The syntax diagram for a **CONST**ant declaration is given in figure 2-5, and the syntax diagram for a constant is expanded in figure 2-6.

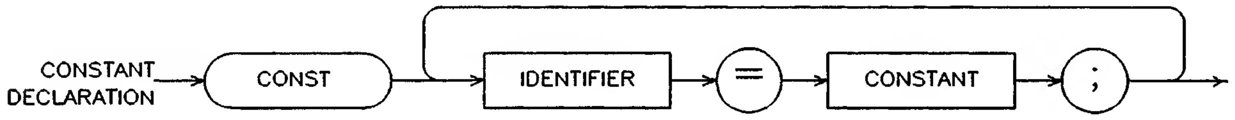


Figure 2-5. CONSTant Declaration Syntax

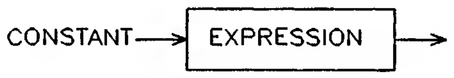


Figure 2-6. Constant Syntax

Constant Expressions

A constant can take the form of a set constructor, the keyword **NIL**, a functional type change, a string literal, a previously defined constant identifier, an integer, a **real** (optionally preceded by + or -), or an expression of an ordinal type i.e., **INTEGER, BOOLEAN, CHAR**acter, subrange, or enumerated type. The use of ordinal expressions as constants is an HP extension.

Example:

```
CONST
    pagesize = 55;
    maxpages = 99;
    pi = 3.14159;
    pagenum = maxpages - pagecount;
    heading_a = 'List is now on.';
```

```
set_const = ['A','B','C'];
unsigned_const = UNSIGNED_16 (1);
```

Constant expressions are constructed according to the rules defined for general expressions (refer to Chapter 4). The operands in an expression must be either literals or constants that have already been defined. In addition, the operands must be ordinal types. No real or set operations are allowed in expressions. The operators allowed are +, -, *, DIV, MOD, and the predefined functions **ORD, CHR, PRED, ABS,** and **SUCC**.

TYPE DEFINITIONS

Types defined in this chapter are the Predefined Types and Structured Types.

Data items can be characterized by their type. The **TYPE** determines a set of attributes as follows:

a. The set of permissable operations that may be performed on an object of that type.

b. The set of values that may be assumed by an object of that type.

c. The amount of storage required by objects of that type.

Certain types are predefined. Other types can be defined by the user.

TYPE DECLARATION

In the type declaration section, an identifier can be associated with a type definition. The reserved word **TYPE** precedes one or more type definitions. A type definition consists of an identifier, the equals (=) sign, and a data type.

The **TYPE** declaration syntax is shown in figure 2-7.

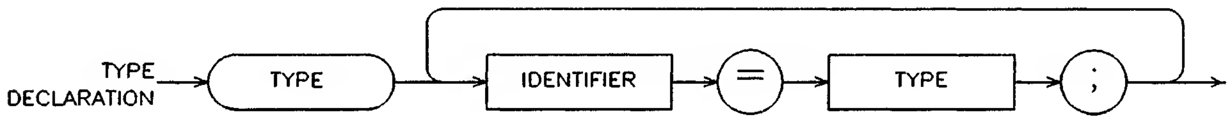


Figure 2-7. TYPE Declaration Syntax

The following paragraphs explain the permissible values and operations for the various data types.

The three most general catagories of data type are simple, structured, and pointer.

Simple data types are the types ordinal, real, or longreal. Ordinal types include the standard types INTEGER, BYTE, SIGNED__8, SIGNED__16, SIGNED__32, UNSIGNED__8, UNSIGNED__16, UNSIGNED__32, CHAR, and BOOLEAN, as well as enumerated and subrange types defined by the user.

Structured data types are the types array, record, set, or file. The standard type TEXT is a variant of the file type.

Pointer data types define pointer variables which point to dynamically allocated variables on the heap.

The **TYPE** syntax is shown in figure 2-8.

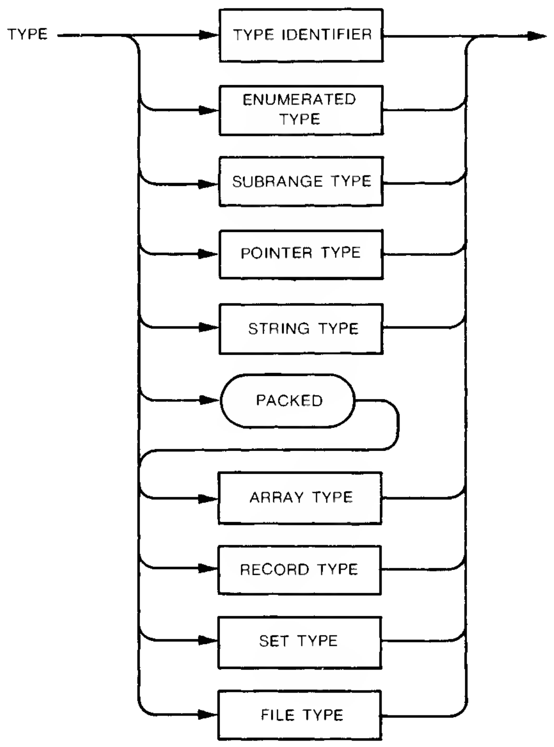


Figure 2-8. TYPE Syntax

SIMPLE TYPES

All simple types define an ordered set of values. Simple types are the types ordinal, real, or longreal.

Ordinal Types

Ordinal types are the predefined ordinal types: **BOOLEAN, CHAR, INTEGER, BYTE, SIGNED__8, SIGNED__16, SIGNED__32, UNSIGNED__8, UNSIGNED__16,** and **UNSIGNED__32**; the user defined ordinal types, enumerated and subrange; and type identifiers that have been equated to another ordinal type.

Predefined Ordinal Types

BOOLEAN. The type Boolean is an ordinal type having two elements, BOOLEAN = (FALSE,TRUE) and occupies one byte of memory. Implicit is the concept that false < true. The operators applicable to Boolean operands are **NOT, AND, OR. NOT** takes precedence over **AND; AND** takes precedence over **OR.** The relational operators always yield Boolean values. The permissible operators are as follows:

assignment	:=
boolean	NOT, AND, OR
relational	<, <=, =, <>, >=, >, IN

CHAR. The type CHAR comprises the ASCII 8-bit character set. Variables of type CHAR occupy one byte of memory. The operators defined for CHAR operands and the predefined functions that result in CHAR values are summarized as follows:

Assignment Operator	:=
Relational Operators	<, <=, =, <>, >=, >, IN
Predefined Functions	CHR, PRED, SUCC

INTEGER. The INTEGER type is predefined as a subrange of the negative and positive Integers. For the 6800 and 6809 microprocessors, INTEGER variables occupy one 16-bit word while for all other processors, INTEGER variables occupy two 16-bit words.

MININT and **MAXINT** are predefined constants which are the values of the minimum and maximum integers for a particular processor.

For the 6800 and 6809 microprocessors, MININT, MAXINT, and INTEGER are defined as follows:

```
CONST
  MININT = -32768;
  MAXINT =  32767;

TYPE
  INTEGER = MININT..MAXINT;
```

For all other microprocessors, MININT, MAXINT, and INTEGER are defined as follows:

```
CONST
  MININT = -2147483648;
  MAXINT =  2147483647;

TYPE
  INTEGER = MININT..MAXINT;
```

The operators defined for integer operands and functions returning INTEGER values are as follows:

Assignment Operator	:=
Relational Operators	<, <=, =, <>, >=, >, IN
Arithmetic Operators	+, -, *, DIV, MOD
Predefined Functions	ABS, LINEPOS, MAXPOS, ORD, POSITION, PRED, ROUND, SUCC, TRUNC

BYTE, SIGNED__8, SIGNED__16, and SIGNED__32. These predefined types are subranges of negative and positive integers. All operations that may be performed on INTEGER types may be performed on these types.

BYTE and SIGNED__8 variables occupy one byte of memory and are defined as follows:

```
TYPE
  BYTE = -128..127;
  SIGNED__8 = -128..127;
```

SIGNED__16 variables occupy two bytes of memory and are defined as follows:

```
TYPE
  SIGNED__16 = -32768..32767;
```

SIGNED__32 variables occupy four bytes of memory. SIGNED__32 is not defined for the 6800 and 6809 microprocessors.

```
TYPE
  SIGNED__32 = -2147483648..2147483647;
```

UNSIGNED__8, UNSIGNED__16, and UNSIGNED 32. These types are defined as a subrange of positive integers. Unsigned types are not compatible with signed types.

Assume for the following definitions, the following type definition:

```
TYPE
  UNSIGNED = {largest unsigned type for processor};
```

UNSIGNED__8 variables occupy one byte of memory and may contain values in the range 0 through 255.

```
TYPE
  UNSIGNED__8 = UNSIGNED(0)..UNSIGNED(0FFH);
```

UNSIGNED__16 variables occupy two bytes of memory and may contain values in the range 0 through 65535.

```
TYPE
  UNSIGNED__16 = UNSIGNED(0)..UNSIGNED(0FFFFH);
```

UNSIGNED__32 variables occupy four bytes of memory and may contain values in the range 0 through 4,294,967,295. UNSIGNED__32 is not defined for the 6800 and 6809 microprocessors.

```
TYPE
  UNSIGNED__32 = UNSIGNED(0)..UNSIGNED(0FFFFFFFFH);
```

The operators defined for unsigned operands and predefined functions that result in unsigned values are as follows:

Assignment Operator	:=
Relational Operators	<, <=, =, <>, >=, >, IN
Arithmetic Operators	+, -, *, DIV, MOD
Predefined Functions	PRED, SUCC

USER DEFINED ORDINAL TYPES

The user defined types discussed in the following paragraphs are the enumerated type and subrange type.

Enumerated Type

Enumeration defines an ordered set of values by listing the identifiers of the ordered values. The identifiers are constants that have ordinal values beginning with 0 for the first identifier, 1 for the second identifier, and so forth. Enumerated type variables occupy one byte of memory. The syntax diagram for the enumerated type is shown in figure 2-9.

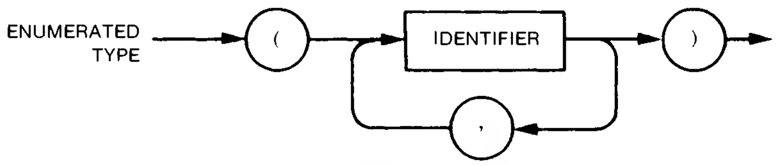


Figure 2-9. Enumerated Type Syntax

Examples:

```
color = (black, brown, red, orange);
day = (sunday, monday, tuesday);
```

The ordinal value of black is 0. The ordinal value of orange is 3. The ordinal value of monday is 1.

The operators defined for enumerated type operands and the operations that result in enumerated type values are as follows:

Assignment Operator	:=
Relational Operators	<, <=, =, <>, >=, >, IN
Predefined Functions	ORD, PRED, SUCC

Subrange Type

A subrange type is a sequential subset of an ordinal base type. A subrange type consists of a lower bound, the special symbol (..), and an upper bound. The upper and lower bounds must be constant values of the same ordinal type. The lower bound cannot be greater than the upper bound.

The amount of memory occupied by subrange variables depends on the values of the upper and lower bounds. In general, the amount of memory will be the smallest that can contain all the values in the subrange. For example:

```
TYPE
  S1 = -128..127; {occupies one byte}
  S2 = -129..128; {occupies two bytes}
```

A variable of a subrange type possesses all of the attributes of the base type except that its values are restricted to the specified closed range. The syntax for the subrange type is shown in figure 2-10.

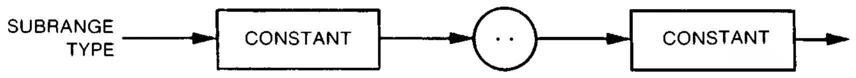


Figure 2-10. Subrange Type Syntax

Examples:

```
TYPE

  Dip = 1..99;
  Alpha = 'A'..'K';
```

REAL TYPES

Real types are the predefined types REAL, LONGREAL, and identifiers that have been equated to real types.

REAL. The set of Real numbers is a subset of whole numbers and is not an ordinal type. The Real number range includes values between +/-10^{38} with six significant decimal digits. Each real number occupies two words in memory. The operators defined for REAL operands and the operations that result in REAL values are as follows:

Assignment Operator	:=
Relational Operators	<, <=, =, <>, >=, >
Arithmetic Operators	+, -, *, /
Predefined Functions	ABS, ARCTAN, COS, EXP, LN, SIN, SQRT

LONGREAL. The set of Longreal numbers is a subset of real numbers. Each longreal number occupies four words in memory. The precision of Longreal is greater than that of Real. The values of the Longreal range are between $+/-10^{308}$ with 15 significant decimal digits. Any of the set of operators applicable to Real numbers are also applicable to Longreal numbers.

STRUCTURED TYPES

The structured types Array, Record, Set, and File are characterized by component type and by the structuring method. A structured type definition may contain an indication of the preferred data representation by use of the term PACKED. The term PACKED is an indication to the compiler that data storage is to be economized.

Array

An array is made up of a fixed number of components, each of which can be directly accessed. Each array has an index by which a component is selected from the array. The index must be an ordinal type, e.g., [1..6]. The number of elements in the array is specified by the index. The components can be any type; but all components are of the same type, called the base type. It is illegal to use the form [INTEGER] as an index, even though INTEGER is an ordinal type. The syntax of the Array is shown in figure 2-11.

Examples:

```
TYPE
   Root = ARRAY ['1'..'6'] OF REAL;
   Freq = PACKED ARRAY ['1'..'6'] OF REAL;
```

Multi-dimensioned arrays, i.e., arrays of arrays, are possible by use of the following format:

```
TYPE
   row = ARRAY [1..5] OF REAL;
   matrix = ARRAY [1..10] OF row;
```

or shortened to the equivalent form:

```
TYPE
   matrix = ARRAY [1..5] OF ARRAY [1..10] OF REAL;
```

or reduced further to the form:

```
TYPE
   matrix = ARRAY [1..5, 1..10] OF REAL;
```

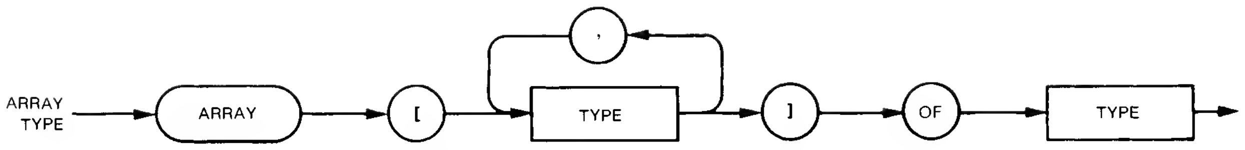


Figure 2-11. Array Syntax

String Data Types

Strings are a family of standard data types that are similar to packed arrays of character, but have special properties. String data types are an HP extension to standard Pascal.

The type STRING is predefined as follows:

```
TYPE
  STRING = PACKED ARRAY[0..225] OF CHAR;
```

Shorter string data types may also be defined. The lower bound of the packed array index type must be zero (0) and the upper bound must be less than or equal to 255. For example:

```
TYPE
  STRING__10 = PACKED ARRAY[0..10] OF CHAR;
```

String data types have a dynamic length. The length is contained in the zero byte position of the packed array. The length may be any value from zero to the maximum declared length of the string variable. For example:

```
VAR
  S: STRING;
  I: INTEGER;
BEGIN
  S := 'ABC'; {Sets S[0] := CHR(3);}
  I := ORD (S[0]); {I now contains 3}
```

String data types of any length are compatible with one another. In addition, string data types are compatible with string literals. The individual characters of a string may be accessed like the elements of any other array.

The operators that may be used with string operands are the following:

Assignment Operator	:=
Relational Operators	<, <=, =, <>, >=, >

Set

A set is the powerset (set of all subsets) of an ordinal type called the base type. The base type may be any ordinal type containing 256 or fewer elements. The syntax of a SET type is shown in figure 2-12.

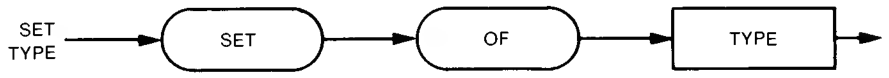


Figure 2-12. Set Type Syntax

The set base type must be an ordinal type. In the case of a subrange of integers, the low bound must be >= to 0 and the high bound must be <= 255.

Relational operators for sets include =, >=, <=, and <>. These operators can be used between sets with results that are Boolean. The symbol IN may be used between an ordinal expression and a simple set expression.

Examples:

```
CHARSET   = SET OF CHAR;
FRUIT     = (apple, banana, cherry, peach, pear, pineapple);
FRUITSET  = SET OF FRUIT;
SOMEFRUIT = SET OF apple..cherry;
```

Sets can be manipulated by set union (+), set difference (-), and set intersection (*) to define new element groups. The maximum number of elements in a set is limited to 256.

Record

RECORD is a Pascal reserved word signifying a structured data type having a fixed number of elements. These elements, called fields, can be of different types. The fields are enumerated and their types defined in the record **TYPE** declaration. Different records may have fields of the same name, but fields within a record must have distinct names. The field list follows each RECORD identifier. Each **RECORD** declaration is completed by END;.

A **RECORD** type definition may contain a "variant" part. This enables variables of type **RECORD**, although of identical type, to exhibit structures that differ in the number and type of their component parts. The "variant" part may contain an optional "tag" field. The value of the tag field indicates which of the variants is currently valid. If a tag field is not specified, then determination of which variant is currently valid is left to the programmer.

Each label in the variant **CASE** declaration must be of the same type as the tag type. Fields of type **FILE** or types which contain files are not permitted in the variant part of a **RECORD**. The label **OTHERWISE** is not allowed in the variant **CASE** declaration. The syntax for a record type is shown in figure 2-13, and the syntax for a field list is shown in figure 2-14.

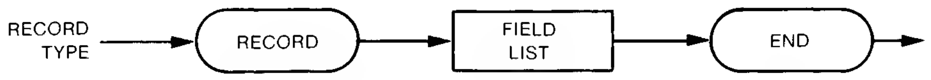


Figure 2-13. Record Type Syntax

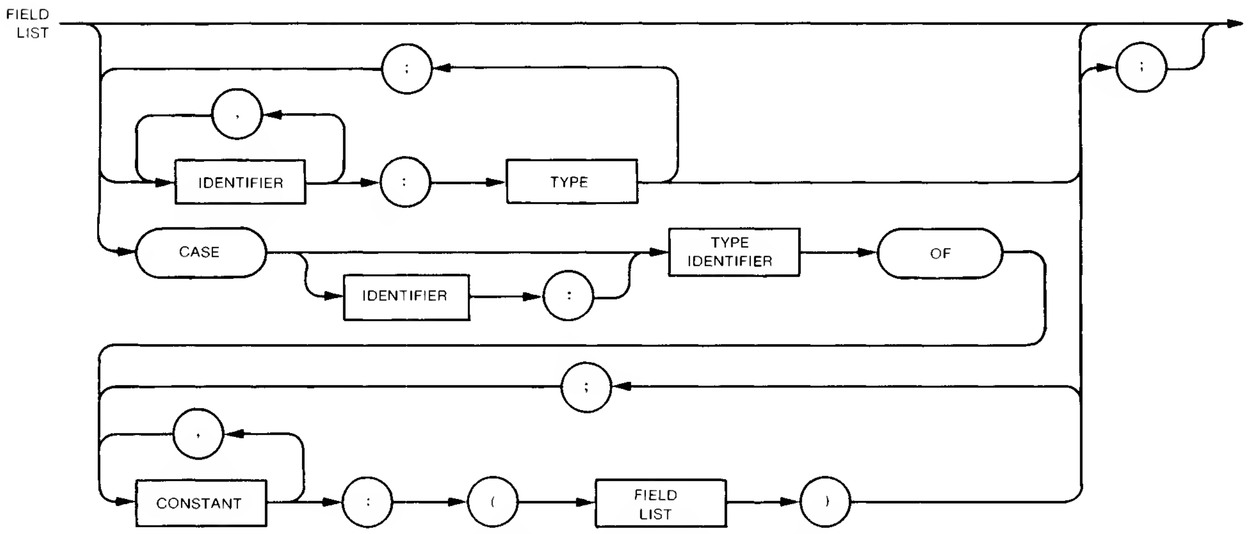


Figure 2-14. Field List Syntax

File

A file type definition specifies a data structure consisting of a sequence of components which are all of the same type. Only one component of a file is accessible at a time. Files are usually associated with peripheral storage devices and their length is not specified in the program. See figure 2-15 for a diagram of the File Type syntax.

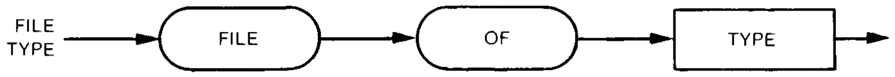


Figure 2-15. File Type Syntax

The component type of a FILE can be any type except FILE or a type which contains a file.

A part of every file variable is the buffer variable. Given the following definition:

```
VAR F : FILE OF T;
```

then the buffer variable, F^, is an ordinary variable of type T. The buffer variable is used to hold the currently accessible component of a file when reading or writing to the file.

POINTER TYPES

Variables that are declared in a program are accessible by their identifiers. These variables exist during the entire execution of the level of program to which they are local, and are therefore called static variables.

Dynamic variables can be generated without any correlation to program structure by using the standard procedure NEW(p). New memory space is allocated for the new dynamic variable, and the pointer variable (p) holds the address of the new dynamic variable. Thus a pointer variable may "point" to a dynamic variable. See figure 2-16 for the pointer type syntax diagram.

A pointer may only refer to dynamic variables of a single type called the "base type". The base type is specified in the pointer definition. A pointer variable may be assigned the value NIL. NIL points to no location in memory. NIL is a reserved word that may be used in a pointer at the end of a linked data structure to indicate the end of the data structure.

The base type identifier is an exception to the rule that all identifiers must be declared before they are used.

Examples:

```
father, mother, child, sibling: ^person;
carbon, film, wirewound: ^resistor;
```

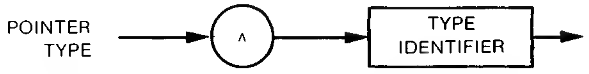


Figure 2-16. Pointer Type Syntax

VARiable

VARiables are locations in memory that are identified by name, and exist during the entire execution of the level of program to which they are local. Variables that have been declared are called static variables.

Dynamic variables, on the other hand, can be generated without any correlation to program structure. Variables contain values that can be changed during the execution of the program. The **VAR**iable declaration syntax is shown in figure 2-17.

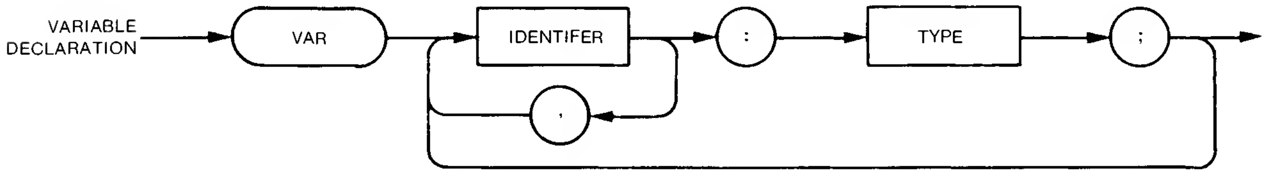


Figure 2-17. VARiable Declaration Syntax

VARiable DECLARATION

A variable declaration associates an identifier with a type. The identifier may then appear as a variable in statements.

The reserved word VAR precedes one or more variable declarations. A variable declaration consists of an identifier, a colon (:), and a type. The user may list any number of identifiers separated by commas. The identifiers will then be variables of the same type.

Examples:

```
VAR

    aset : char;
    bset, dset, flop : INTEGER;
    freq : PACKED ARRAY [1..15] of REAL;
    root : ARRAY [(alpha, beta)] of COLOR;
    cset : FILE OF CHAR;
    nset : SET OF noun;
    p1, p2 : ^person;
```

The value of a variable is undefined at the time of declaration.

ROUTINE DECLARATIONS

PROCEDURE and **FUNCTION** declarations may take place in the declaration portion of the main program, or within other procedures or functions. Routines must be declared before they are used. Each routine, whether procedure or function, is declared in a similar fashion. The routine heading is followed by either the directives **FORWARD** or **EXTERNAL** or by a block that contains the declarations and statements comprising the routine. The routine declaration syntax is shown in figure 2-18.

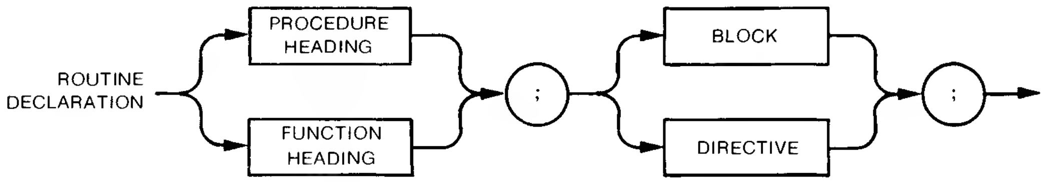


Figure 2-18. Routine Declaration Syntax

Procedure Declaration

Procedures perform specific tasks or algorithms by execution of the statements within the procedure. Each procedure must be declared before its use. The procedure heading syntax is shown in figure 2-19.

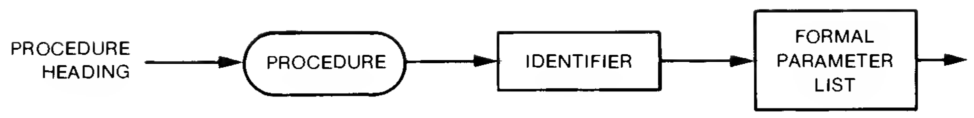


Figure 2-19. PROCEDURE Heading Syntax

Example:

```
PROGRAM REG21;
   VAR  I,N  : INTEGER;
      X,Y  : REAL;
   PROCEDURE SWAP (VAR P,Q :REAL);
   VAR  TEMP : REAL;
   BEGIN
      TEMP := P;
      P    := Q;
      Q    := TEMP
   END;
BEGIN
   .
   .
END.
```

Function Declaration

The function declaration consists of a heading and a main block. The function heading consists of the function identifier, a formal parameter list, and the type of function result. The type of the function result can be any type except a file, or a type containing a file. Within the function main block at least one statement must assign a value to the function identifier. See figure 2-20 for the FUNCTION heading syntax.

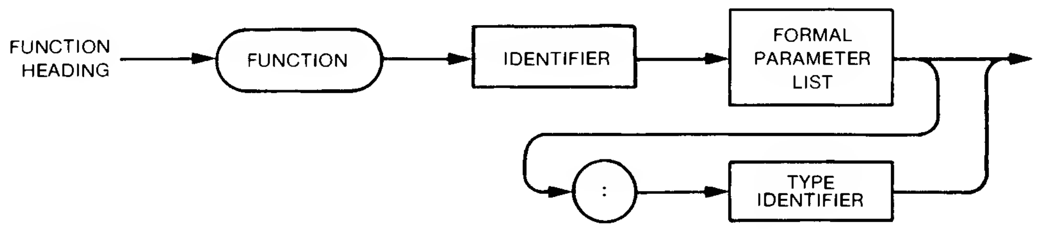


Figure 2-20. FUNCTION Heading Syntax

Functions perform specific tasks or algorithms by execution of statements within the **FUNCTION**. The function is further identified by a type; and the value generated by the function must be of that type, and assignable to the function identifier.

Example:

```
{function declaration}

FUNCTION Sqrt (x:REAL):REAL;
CONST eps = 1E-5;
VAR X0, X1:REAL;
BEGIN X1 :=X; {X >1, Newton's method}
   REPEAT X0 := X1; X1 := (X / X0 + X0) * 0.5
   UNTIL abs (X1-X0) < eps * X1;
   Sqrt := X0
END;

{function call}

    BEGIN  {start of program}
    . . .
      .
      y := sqrt (3.5);
      .
      .
      .
      .
    END. {end of program}
```

PARAMETER LISTS

Formal Parameter List

The formal parameters for both functions and procedures can be value parameters or variable parameters. The syntax diagram of the formal parameter list is shown in figure 2-21.

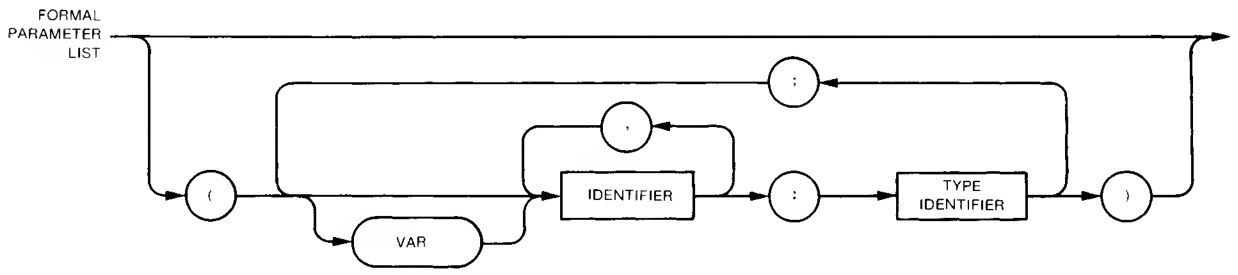


Figure 2-21. Formal Parameter List Syntax

Actual parameters are the values used in the execution of procedures and functions. The syntax diagram of the actual parameter list is shown in figure 2-22.

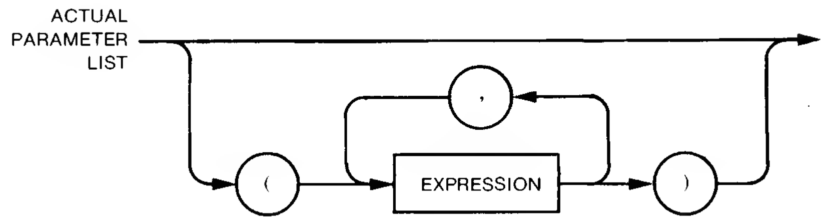


Figure 2-22. Actual Parameter List Syntax

Value Parameter

The actual parameter must be an expression, i.e., something found on the right side of an assignment statement. The corresponding formal parameter represents a local variable in the called routine, and the current value of the expression is initially assigned to this variable. Actual value parameters must be assignment compatible with the type of the corresponding formal parameter.

Variable Parameter

The actual parameter must be a variable, and the corresponding formal parameter represents the actual variable during the entire execution of the routine.

An actual variable parameter must have the same type as the corresponding formal parameter.

DECLARATIONS WITHIN ROUTINES

The declaration part of a procedure or function contains the declarations of constants, types, labels, variables, and other routines. These declarations are local to the routine in which they are declared. Declarations within routines take the same form as the program declaration.

Routine Body

The body of a routine is a compound statement that describes the execution of the routine toward an end result. The result of a **FUNCTION** is a value. The result of a **PROCEDURE** is an action. The syntax for the routine block is the same as that of a main program block.

Directives

All routines must be declared before they are called. If the routine's block does not immediately follow the routine heading, then a **FORWARD** directive must be used to inform the compiler of the location of the block. A **FORWARD** declaration is composed of the routine heading, including the parameter list if used, the function result type if applicable, followed by the directive. The routine must be fully declared before the end of the current scope. The parameter list, and result type for a **FUNCTION**, may not be respecified.

Example:

```
FUNCTION exclusive__or (x,y:boolean) : boolean;
   FORWARD;
      .
      .
      .
  FUNCTION exclusive__or;
     BEGIN
        exclusive__or := (x and not y) or (not x and y)
     END;
```

The directive EXTERNAL indicates that a procedure or function is not defined in the current program. Rather, the routine exists in another Pascal program or assembly language module. The module containing the actual definition of the external routine must be linked to the present program before the present program can be executed.

Example:

```
PROGRAM A;

VAR I : INTEGER;

FUNCTION MIN(A,B : INTEGER): INTEGER; EXTERNAL;

BEGIN
I := MIN (5,10);
END.
```

```
PROGRAM B;
$EXTENSIONS ON$
$GLOBPROC ON$
FUNCTION MIN(A,B : INTEGER):INTEGER;

BEGIN
IF A <= B THEN
   MIN := A
ELSE
   MIN := B;
END;
.
```

Recursive Routines

A routine that calls itself is a recursive routine. Use of the routine identifier within the routine body indicates recursive execution of the routine. If a **FUNCTION** identifier appears on the left of an assignment statement, however, only the assignment is executed. It is also possible for a first routine to call a second routine in which the first routine is called. That action is an indirect recursion.

SCOPE

Certain objects in PASCAL/64000 programming have a related scope of utility. Those objects are:

a. labels
b. constants
c. types
d. variables
e. formal parameters
f. routines

The scope of an object pertains to the level of the program in which the object is declared or defined. Within a routine declaration, the declaration part specifies local labels, constants, types, variables, and routines. Execution of the routine may access labels, variables, constants, types, and parameters declared at the same or outer levels of declaration. No outer level program execution, however, can access an inner level identifier. In the case of two identifiers having different scopes but having the same spelling, the outer identifier will be inaccessible to the inner identifier. No two identifiers having the same scope can have the same spelling.

Chapter 3

STATEMENTS

GENERAL INFORMATION

A statement is a sequence of special symbols, reserved words, and expressions that either performs a specific set of actions on data or controls program flow. A list of the statement types along with a brief description of each follows (refer to figure 3-1 for statement syntax):

STATEMENT TYPE	PURPOSE
compound	group statements
empty	do nothing
assignment	assign a value to a variable
procedure	activate a procedure
GOTO	transfer control unconditionally
IF, CASE	conditional selection
WHILE, REPEAT, FOR	repeat a group of statements
WITH	manipulate record fields

Empty, assignment, procedure, and **GOTO** statements are simple statements. **IF, CASE, WHILE, REPEAT, FOR,** and **WITH** statements are structured statements because they may contain other statements.

A statement label may be associated with any statement in a program body. The label must be assigned in the declaration section of the block and is used by the **GOTO** statement for branching purposes.

The following paragraphs describe each type of statement.

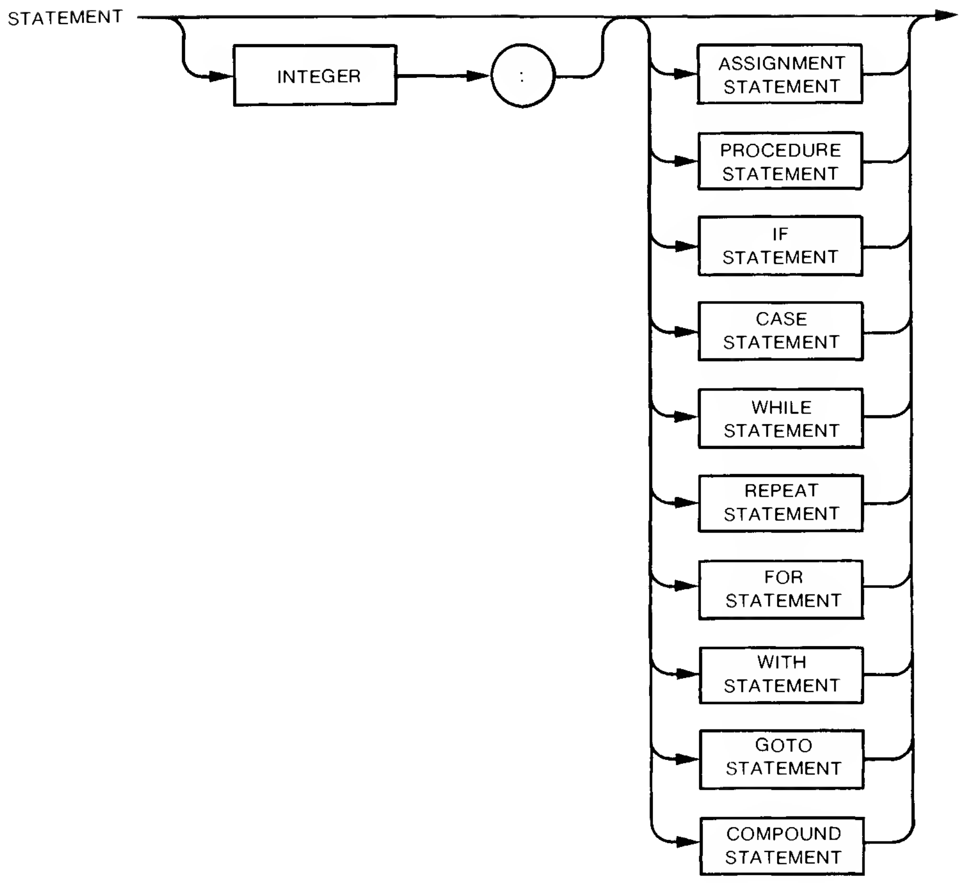


Figure 3-1. Statement Syntax

COMPOUND STATEMENT

The compound statement is used as a means of treating a group of statements as a single statement. The compound statement is delimited by the reserved words **BEGIN** and **END**. The statements enclosed by **BEGIN** and **END** are executed in the order written. The compound statement has two primary uses:

a. As the body of a procedure, function, or program;

b. As a structured statement that may contain other statements. Usually where a substatement is allowed, the default is only one statement. The compound statement is useful if more than one statement is to be executed.

Compound statements may be used as part of **IF, CASE, WHILE, REPEAT, FOR,** and **WITH** statements. Delimiters are required with each of the compound statements. The **BEGIN/END** pair is used in all cases except **REPEAT** and **CASE** statements.

REPEAT/UNTIL delimit the **REPEAT** statement, and **CASE/END** delimit the **CASE** statement. The compound statement syntax is shown in figure 3-2.

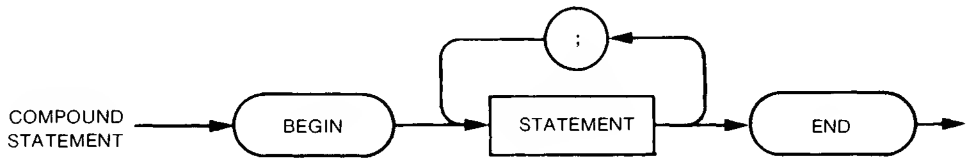


Figure 3-2. Compound Statement Syntax

EMPTY STATEMENT

The empty statement is denoted by no symbol and performs no action. It is used to indicate that no action is to be taken as the result of a condition evaluation.

Example:

```
IF a < b
   THEN
   ELSE
      GOTO 1027
```

THEN has no action statement associated with it, therefore an empty statement exists.

ASSIGNMENT STATEMENT

The assignment statement is used to change the value of a variable. The variable can be of any type except a file type, or a structure containing a file. The type of the variable and the type of the expression must be assignment compatible; e.g., a variable of type **INTEGER** cannot be assigned a value of type **CHAR**. The identifier on the left side of the assignment symbol may be either a variable identifier, a field identifier, or a function identifier. If the identifier is a **FUNCTION** identifier, the assignment statement must be made within the block of the **FUNCTION**. The assignment statement syntax is shown in figure 3-3.

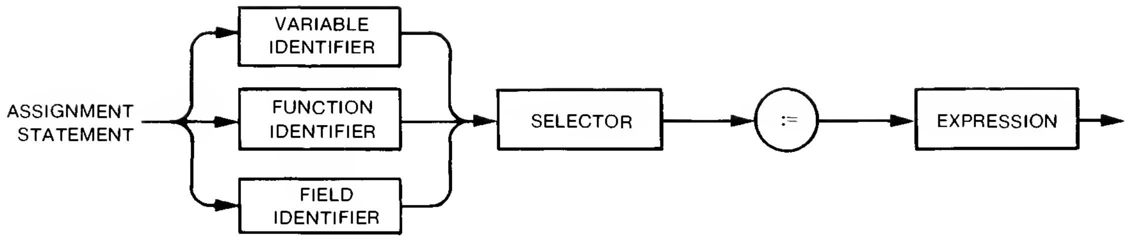


Figure 3-3. Assignment Statement Syntax

Example:

```
fctr := 31.25;
flop := flim * fctr;
```

PROCEDURE STATEMENT

The procedure statement transfers program execution to a procedure. Upon completion of the procedure, program execution is transferred to the statement that follows the procedure statement. The procedure identifier must be the name of either a predefined procedure or a procedure declared previously in a procedure declaration. If the formal declaration of the procedure includes a parameter list, the procedure statement must have the actual parameters. The actual parameter list must agree in number, order, and type with the formal parameter list. The procedure statement is illustrated in the following examples, and the syntax is shown in figure 3-4.

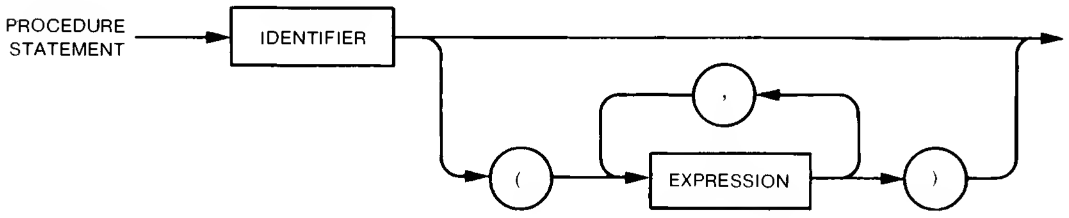


Figure 3-4. Procedure Statement Syntax

Example:

```
PROCEDURE freqgen (VAR fctr,rctr:integer); {procedure declaration}

   BEGIN
      .
      .
      .
   END;  {procedure declaration}

BEGIN  {program}
   .
   .
   .
   freqgen (apron,ramp);
   .
   .
   .
END.  {program}
```

GOTO STATEMENT

The **GOTO** statement is used in conjunction with a label. The label must be an integer in the range of 1..9999. Program execution is transferred to the statement named by the label. The label in a **GOTO** statement must be defined in the same body as the **GOTO** statement. The **GOTO** statement syntax is illustrated in figure 3-5.

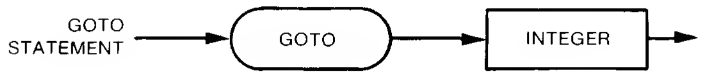


Figure 3-5. GOTO Statement Syntax

In Pascal/64000 the **GOTO** statement should not direct program execution into the middle of any **FOR** or **WITH** statement because results may be undefined.

IF STATEMENT

The **IF** statement chooses one of two possible responses, based on a given condition. The two responses possible are **THEN** and **ELSE**. The expression that follows **IF** must be a boolean type. When the **IF** statement is executed, the expression is evaluated to be either **TRUE** or **FALSE**. If the value is true, the action following **THEN** is performed. If the value is false, the action following **ELSE** is performed. If the value is false and no **ELSE** action is specified, no action is taken. By implication, however, the remainder of the program becomes the **ELSE** action. **ELSE** parts that appear to belong to more than one **IF** statement are always associated with the nearest **IF** statement. Note that a semicolon may not separate a **THEN** statement from the related **ELSE** statement. The **IF** statement syntax is illustrated in figure 3-6.

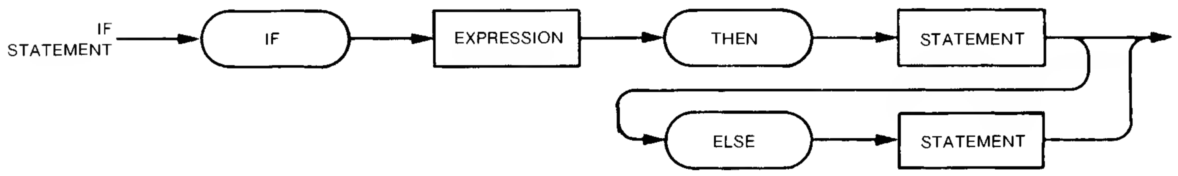


Figure 3-6. IF Statement Syntax

CASE STATEMENT

The **CASE** statement, like the **IF** statement, is used to select a certain action based upon the value of an expression. The **CASE** statement, however, can select from more than two courses of action. If none of those courses of action are selected an **OTHERWISE** statement is executed. The **OTHERWISE** portion of the **CASE** statement is an HP extension of standard Pascal. **CASE** statement syntax is illustrated in figure 3-7. The **CASE** expression may be any ordinal type, including boolean, integer, character, and user-defined enumeration and subrange types. The expression, called the selector, is used to choose which statement is to be executed. Each constant expression in the list of labels must be compatible with the type of the selector. A label may only appear in one list. The statement associated with the label list containing the value matching the selector is executed. The statement associated with the **OTHERWISE** part is executed if the selector does not match any of the labels. Specifically, when a **CASE** statement is executed:

a. The selector expression is evaluated.

b. If the value appears in a label list within the **CASE** statement, the statement associated with that list is executed and main program execution continues with the statement following the **CASE** statement.

c. If the value does not appear in any label list the statements appearing between **OTHERWISE** and **END** are executed, and program execution resumes with the statement following the **CASE** statement.

d. If the value does not appear in any label list and no **OTHERWISE** clause exists, the result will be a run time error.

CASE statement values are restricted to those that can be expressed in the range of -2^{15} (-32768) through $+2^{15}-1$ (+32767) for signed types; or 0 through $2^{16}-1$ (+65535) for unsigned types.

CASE statements may be nested to any level.

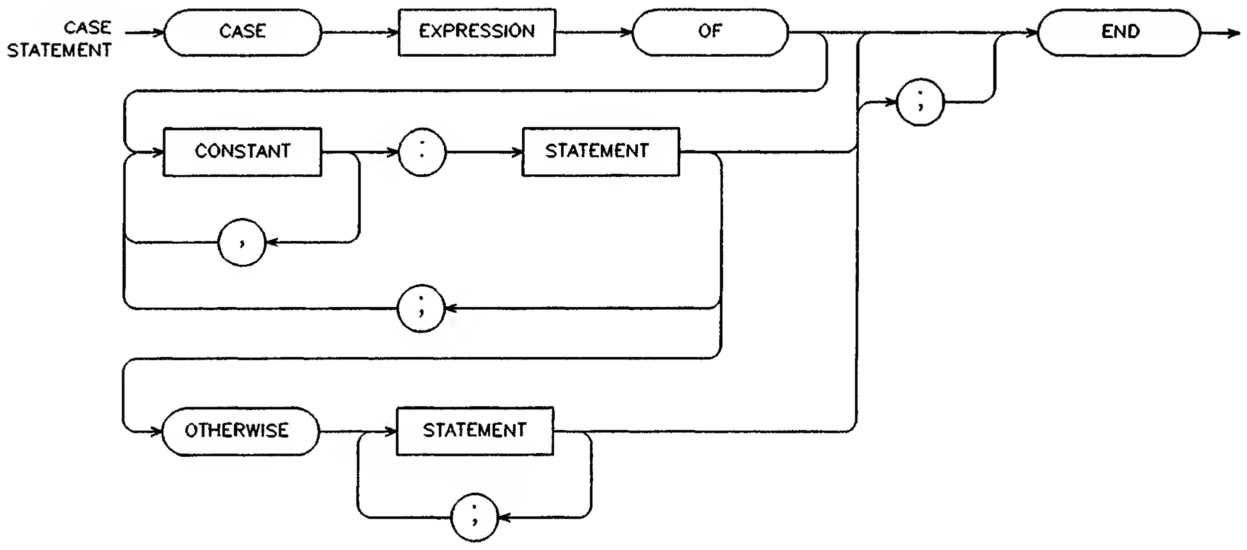


Figure 3-7. CASE Statement Syntax

WHILE STATEMENT

The **WHILE** statement is a repeating statement used to execute an action so long as a given expression is true. The expression is evaluated before execution, in contrast to the **REPEAT** statement which is evaluated after execution. The expression must be of the boolean type. Each time the evaluation is true the **WHILE** statement is executed. When the evaluation becomes false the statement following the **WHILE** statement is executed and program action is continued.

It is necessary that execution of a **WHILE** statement causes a change in data such that the evaluation result becomes false. Otherwise, the **WHILE** statement is never exited, an endless loop exists and execution of the program is never concluded. The **WHILE** expression syntax is illustrated in figure 3-8.

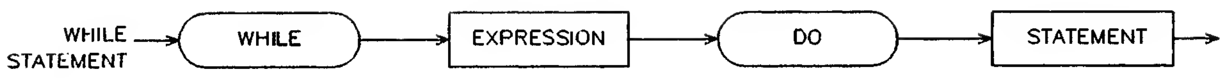


Figure 3-8. WHILE Statement Syntax

REPEAT STATEMENT

The **REPEAT** statement is executed so long as the **UNTIL** Boolean expression is false. The expression is evaluated after execution of the statement enclosed by the **REPEAT/UNTIL** delimiters, in contrast to the WHILE expression which is evaluated before execution. The expression must be of the boolean type.

Each time the evaluation returns false the **REPEAT** statement is executed. When the evaluation returns true the statement following the **REPEAT** is executed and program action is continued. It is necessary that execution of a **REPEAT** statement causes a change in data such that evaluation results in a true value. Otherwise the **REPEAT** statement is never exited, an endless loop exists and execution of the program is never concluded. The **REPEAT** statement syntax is illustrated in figure 3-9.

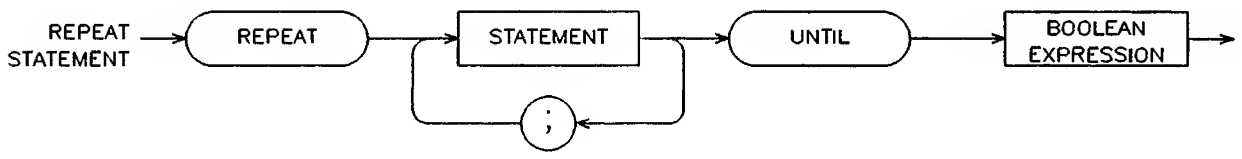


Figure 3-9. REPEAT Statement Syntax

FOR STATEMENT

The **FOR** statement executes a statement once for each value in a range specified by initial and final expressions. A variable, called the control variable, is assigned each value of the range before the corresponding iteration of the statement. The control variable must be a local or global variable, and it also must be an entire variable. In addition, the control variable may be a local formal value parameter, but may not be a formal variable parameter.

The range of values assumed by the control variable is specified, typically:

```
FOR n := 1 TO 10 DO
```

The range specified is 1 TO 10. The **FOR** statement is not executed, and the control variable is not changed if the initial expression is greater than the final expression with the **FOR..TO** statement (or less than the final expression with the **FOR..DOWNTO** statement). The range expressions must be assignment compatible with the type of the control variable. These expressions are evaluated only once, before any assignment is made to the control variable. The **FOR** statement syntax is illustrated in figure 3-10.

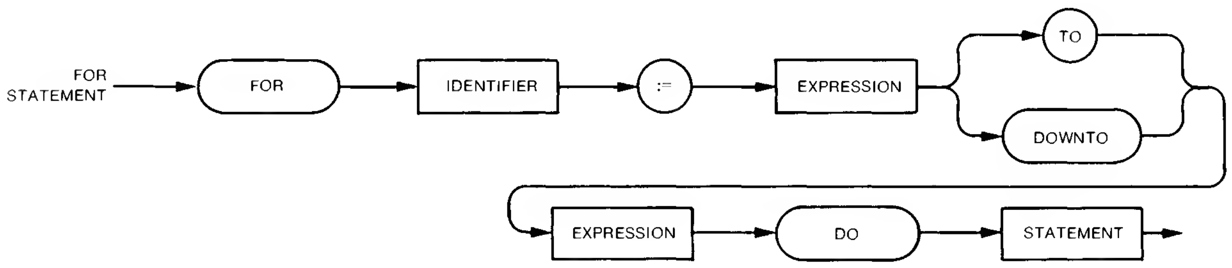


Figure 3-10. FOR Statement Syntax

WITH STATEMENT

The **WITH** statement allows access to record fields without naming the record variable. Within the **WITH** statement any field of any record in the list may be accessed by using only its field name (instead of the normal field selection notation using the period between the record and the field name).

Example:

```
TYPE
   R = RECORD
       aa:INTEGER;
       b,c:REAL;
       END;
VAR
  V : R;
BEGIN
WITH V DO
  aa := 0;
```

WITH statement syntax is illustrated in figure 3-11.

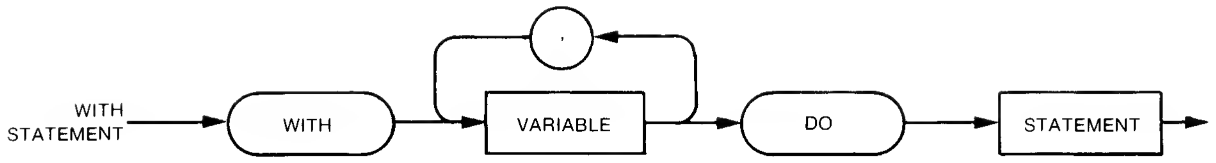


Figure 3-11. WITH Statement Syntax

Chapter 4

EXPRESSIONS

GENERAL INFORMATION

An expression is a construct composed of operators and operands, and is used to compute a value of some type. An operator defines an action to be performed on its operands. Operands denote the objects that operators will use in obtaining a value. An operand may be a literal, a constant identifier, a variable, or it may be a reference to a function. The syntax diagram for expressions is shown in figure 4-1, and is expanded into greater detail in figures 4-2 thru 4-4.

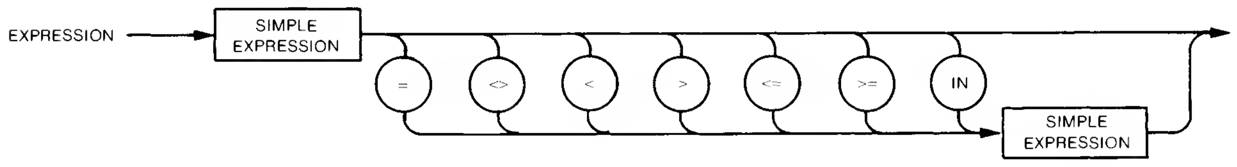


Figure 4-1. Expression Syntax

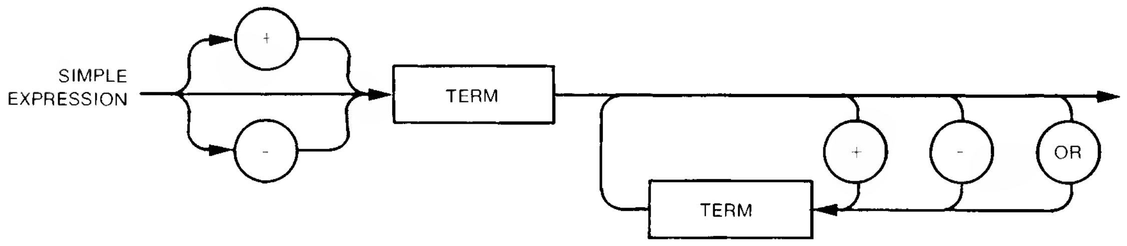


Figure 4-2. Simple Expression Syntax

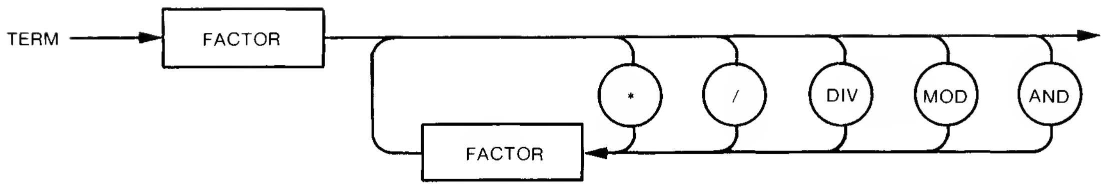


Figure 4-3. Term Syntax

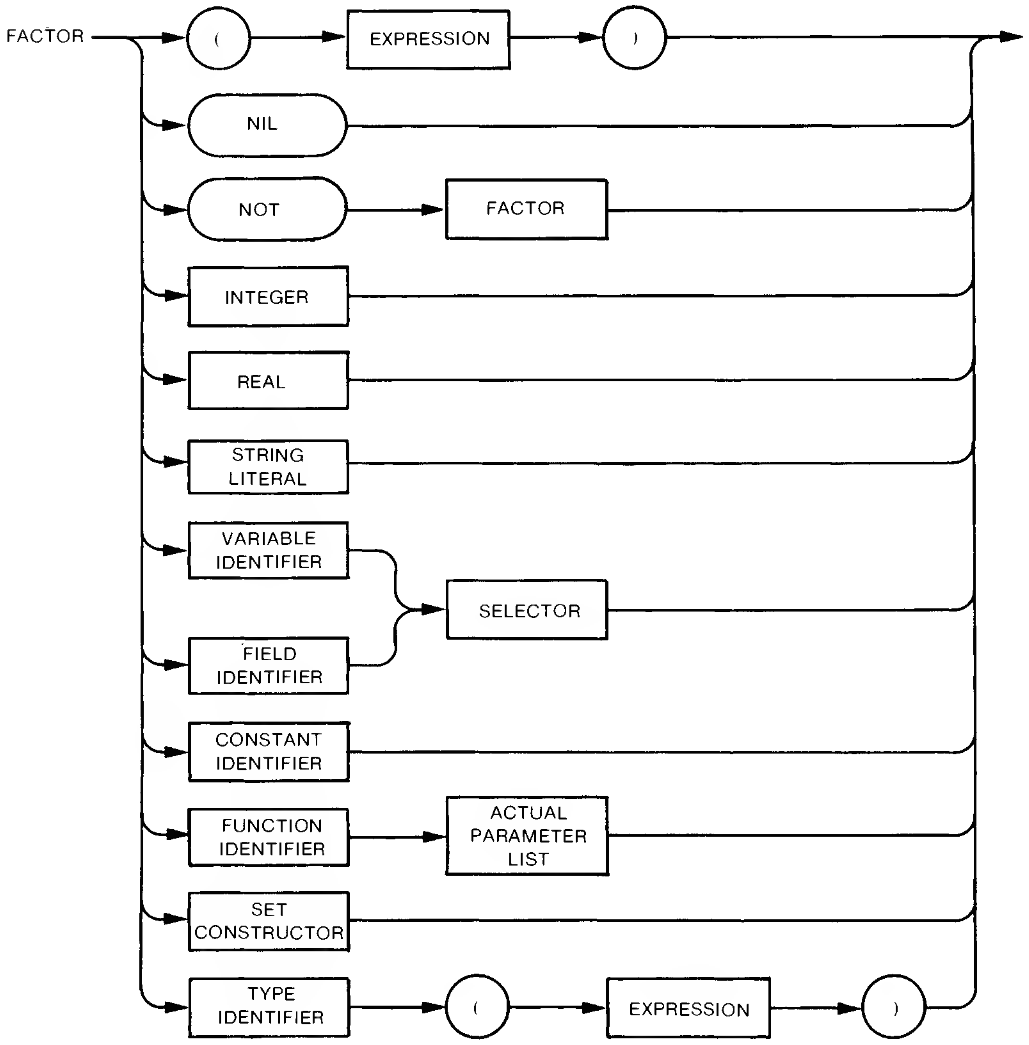


Figure 4-4. Factor Syntax

An expression's type is known when it is written, and never changes. An expression's value, however, may not be known until the expression is evaluated and may be different for each evaluation.

Word, integer, and byte variables may be mixed within expressions. If both operands of a binary operation are of type byte, the compiler performs a byte operation. If one is byte and the other is integer, the compiler first converts the byte value into integer then performs an integer operation. The only time the compiler will automatically truncate an integer to a byte is if the left hand side of an assignment statement is of type byte and the right hand side is of type integer. The integer expression is evaluted and the 16- or 32-bit result is truncated to an 8-bit result before assigning the byte value.

The compiler does not perform run-time checks to ensure that assignments to scalar or subrange variables are in the user defined subrange unless the compiler directive **$RANGE ON$** is in effect.

OPERATORS

Operators are used within expressions to specify certain actions on one or more operands, and to create a new value. The value is determined by the operator, its operands, and the definition of the effect of the operator. With each operator is associated the following:

a. number, order, and type of operands
b. result type
c. precedence

Operator precedence is used to determine the order of element evaluation in an expression. The higher precedence operators are evaluated first. Grouping of operators can alter the precedence value of the group; however, precedence within the group still follows the rules of precedence. The following list shows operators with their order of precedence, from highest to lowest.

NOT
*, /, DIV, MOD, AND
+, -, OR
<, <=, <>, =, >=, >, IN

Operators may either be predefined or user-defined. Predefined operators are the arithmetic, boolean, set, string, and relational operators, and the predefined functions. User-defined operators are references to user-written functions, routines that compute and return a value. The value resulting from any operation may in turn be used as an operand for another operator.

Table 4-1 contains the predefined operators and their meaning.

Arithmetic Operators

Arithmetic operators take numeric operands and produce a numeric result. A numeric type is the type **REAL, LONGREAL, INTEGER, BYTE, SIGNED__8, SIGNED__16, SIGNED__32, UNSIGNED__8, UNSIGNED__16, UNSIGNED__32,** or subranges of signed and unsigned types. Operands of different types may be mixed in an arithmetic operation. In general, the compiler converts the smaller type to the larger type before the operation is performed. The type of the result is the type of the larger type.

Operands for both DIV and MOD must be signed or unsigned integers.

Signed and unsigned arithmetic is supported by all Pascal/64000 compilers. Signed and unsigned arithmetic may not be mixed in the same expression without the use of the functional type change, i.e., for a binary operator, the left operand and the right operand must be signed operands or they must both be unsigned operands. Arithmetic constants appearing in the source text are treated as signed values unless the user specifies that they be treated as unsigned by surrounding them with functional type change operations.

When converting a signed operand to a physically larger signed operand, the smaller signed operand will be sign extended by the compiler. When converting an unsigned operand to a physically larger unsigned operand, the smaller unsigned operand is extended with zeroes.

Integer division (DIV) calculates the truncated quotient of two integers. The sign of the result is positive if both operands have the same sign, and negative if the operands have opposite signs.

A div B is equivalent to trunc (A/B).

The MOD operation produces the remainder of a divide operation.

```
A MOD B = A - ((A DIV B) * B)
```

The operators +, -, *, and / permit operands with different numeric types. Each numeric type has a conversion ranking as follows:

Type	**Rank**
LONGREAL	4 (highest)
REAL	3
INTEGER, SIGNED__32, UNSIGNED__32	2
SIGNED__16, UNSIGNED__16	1
BYTE, SIGNED__8, UNSIGNED__8	0

Table 4-1. Pascal/64000 Operators

Operator	Meaning
+	Numeric UNARY PLUS and ADDITION; set UNION
-	Numeric UNARY MINUS and SUBTRACTION; set DIFFERENCE
*	Numeric MULTIPLICATION; set INTERSECTION
/	REAL DIVISION
DIV	Integer DIVISION
MOD	Integer MODULUS
AND	Logical AND
OR	Logical INCLUSIVE OR
NOT	Logical NEGATION
<	Numeric, string, enumeration LESS THAN
<=	Numeric, string, enumeration LESS THAN OR EQUAL; set SUBSET
=	Numeric, string, enumeration, set, pointer EQUALITY
<>	Numeric, string, enumeration, set, pointer INEQUALITY
>=	Numeric, string, enumeration GREATER THAN OR EQUAL; set SUPERSET
>	Numeric, string, enumeration GREATER THAN
IN	Set MEMBERSHIP

If two operands associated with an operator are not the same rank, the compiler converts the lower to the higher prior to the operation. The result value will be of the same type as the higher ranked operand. To summarize:

One Operand Type	Other Operand Type	Result Type
INTEGER	REAL	REAL
INTEGER	LONGREAL	LONGREAL
REAL	LONGREAL	LONGREAL
SIGNED__8	SIGNED__16	SIGNED__16
UNSIGNED__8	UNSIGNED__16	UNSIGNED__16

Boolean Operators

The Boolean operators perform logical functions on Boolean operands. The Boolean operators are: NOT, AND, OR.

NOT. The **NOT** operator takes one Boolean operand and produces a Boolean result equal to the inverse of the operand.

AND. The **AND** operator yields a Boolean result of true only if all operands are true.

OR. The **OR** operator yields a Boolean result of true if any one of the operands is true; or yields a Boolean result of false only if all of the operands are false.

The AND (OR) operator is used to perform the logical AND (inclusive OR) operation on two Boolean operands. The result is a Boolean value defined in the truth table that follows.

a	b	NOT a	a AND b	a OR b
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

Relational operators with Boolean operands are always fully evaluated. NOT, AND, OR cannot be used on operands of non-Boolean types.

Set Operators

Three set operators are defined which manipulate two expressions having compatible set types and result in a third set. The set operators are: set union (+), set difference (-), and set intersection (*).

SET UNION. The union operator creates a set whose members are all of those elements present in the left set plus those in the right set. Simply, the combining together of two sets into one set.

SET DIFFERENCE. The difference operator creates a set whose members are those elements that are members of the left set but are not members of the right set.

SET INTERSECTION. The intersection operator creates a set whose members are all of those members present in both sets.

The two operands of a set operator must be compatible. Two sets are compatible if they both have the same base type and if their lower and upper bounds are equal. The compiler does not permit operations on sets of different sizes even if the two sets have the same base type.

RELATIONAL OPERATORS

Relational operators are used to compare two operands and return a boolean result. The operands may be ordinal types, reals, sets, pointers, strings, arrays, or records. Relational operators appear between two expressions, that must be compatible, and always result in a value of type boolean. The relational operators are:

<	(less than)
<=	(less than or equal)
=	(equal)
<>	(not equal)
>=	(greater than or equal)
>	(greater than)
IN	(set membership)

Ordinal Relationals

The relationals that can be used with operands of ordinal or subrange type are: <, >, <=, =, <>, and >=. These operators carry the normal definition of ordering for numeric types, and CHAR relationals are defined by the ASCII collating sequence. The order of enumerated constants is defined by the order in which the constant identifiers are listed in the **TYPE** definition. The predefinition of boolean is: BOOLEAN = (FALSE, TRUE) and means FALSE < TRUE. An expression having an ordinal type may also appear as the first operand of the **IN** operator. Some boolean functions may be performed using the relational operators with boolean operands, as shown in the following truth table:

a	b	a<b	a<=b	a=b	a<>b	a>=b	a>b
T	T	F	T	T	F	T	F
T	F	F	F	F	T	T	T
F	T	T	T	F	T	F	F
F	F	F	T	T	F	T	F

<= is the implication operator, = is the equivalence operator, and <> is an exclusive OR operator.

Real Comparison

REAL and LONGREAL operands may be compared using the relationals <, >, <=, >=, =, and <>. REAL operands may also be compared to signed and unsigned integer types. The compiler converts smaller data types to larger data types before the comparison is performed.

String Comparison

Strings may be compared using the operators =, <>, <, <=, >, or >=. The types of operands that may be compared are STRING, PACKED ARRAY [0..n] OF CHAR, or string literals.

A string expression can be compared to any other string expression, including string literals. Strings are compared character by character until a pair of unequal characters are found or until all the characters in the shorter string are used. If two characters are unequal, the ordering of the string is determined by the ASCII collating sequence. If all the characters in two strings are equal up to the length of the shorter string, then the longer string is greater than the shorter string. Two strings are equal only if they have the same length and all characters contained in that length are equal.

Pointer Relationals

Pointers can only be compared using the relationals = and <>. Two pointers are equal if they point to exactly the same object, and are not equal otherwise. Pointers of any type may be compared to the constant **NIL**. Pointers can only be compared to other pointers, and their two pointer types must be identical.

Set Relationals

Two sets can be compared for equality with = and <>. In addition, the <= operator is used to denote the subset operation, and >= denotes the superset operation. One set is a subset of a second set if every element in the first set is also a member of the second set. Also, the second set is said to be a superset of the first set. The < and > operators are not allowed on sets.

The **IN** operator is used to determine whether or not an element is a member of a set. The right operand has the type SET OF T, and the left operand has an ordinal type compatible with T. To test the negative of the **IN** operator, the following form is used: NOT (element IN set).

Array and Record Comparison

Arrays and records can be compared for equality using the relationals = and <>. Two arrays or records are equal if every byte in one structure is equal to every byte in the other structure. Comparison of arrays and records is a Pascal/64000 extension of standard Pascal.

For some microprocessors, the compiler forces certain data types to be aligned on word boundaries. When this is done with a structured data type, there are "holes" or unused data bytes within the structure. The compiler produces warning number 508 in this case. Comparison of these structures may not work because these unused bytes may contain random values.

OPERANDS

An operand can be a literal, a variable, a declared constant, a set constructor, a function call, a dereferenced pointer, or the value of another expression that may be acted upon by an operator to produce a value.

SET CONSTRUCTOR

The set constructor designates one or more values as members of a set whose type may or may not have been previously declared. A set constructor consists of an optional set type identifier and one or more ordinal expressions in square brackets ([..]). See figure 4-5 for the Set Constructor Syntax diagram.

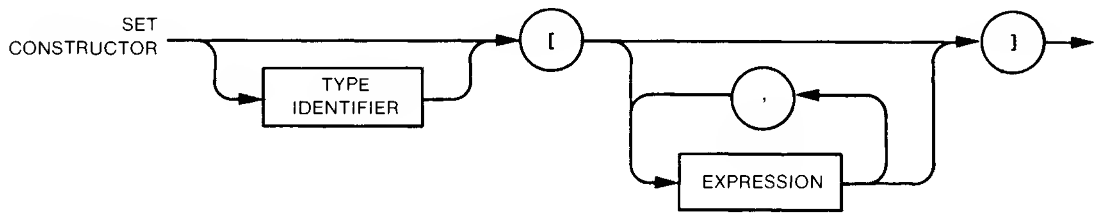


Figure 4-5. Set Constructor Syntax

If no type identifier is used, one of three possible results will occur depending on the type *T* of the elements in the set:

a. If *T* is an integer, then the set created is of type SET OF 0..255. Compile-time checks are performed to ensure that constant set elements are in this range. Thus the set [25,0,255] is legal, but the set [-10, 256] is not legal.

b. If *T* is any other ordinal type, the set created is a set whose base type is the entire ordinal type. The set ['A','T'] has the type SET OF CHARacter.

c. If the empty set ([]) is specified, the type of the set will be determined from context.

The **TYPE** identifier is needed to construct integer sets other than the range 0..255. But it is also desirable to specify the type for sets over other subrange types for efficiency reasons. The set UPPER__CASE ['A'..'T'] requires much less storage than the set ['A'..'T'], and has a corresponding savings in run time.

FUNCTION CALL

A function call activates the block of a standard or declared function. The function returns a value to the calling point of the program. A call to a function, therefore, can be thought of as an operator whose operands are the actual parameters passed to the function; or as an operand whose value is determined by the process in the function.

A function call consists of a function identifier and an optional list of actual parameters in parentheses. See figure 4-6 for a syntactical diagram of a function call.

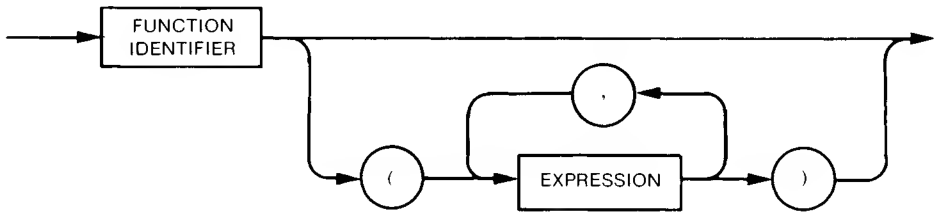


Figure 4-6. Function Call Syntax

The result, whose type is defined in the function heading, is treated identically to the result of any other operator, and may be used inside an expression. Actual parameters must match the function's formal parameters in number, order, and type. If the function's type is structured, then the value returned by the function must be put into a variable before elements of the structure may be accessed.

Functions may be recursive.

SELECTORS

Array Subscripts

Array and string components are selected using subscripts, denoted by square ([]) brackets, and an expression. The subscript, or index, type must be compatible with the expression type appearing in the array type definition. The values of constants and non-constants are checked at run time to make sure those values lie in the range specified in the index type, unless the **RANGE** compiler option is turned **OFF**. The array selector appearing before the brackets may itself be a selected variable. See figure 4-7 for a diagram of the selector syntax.

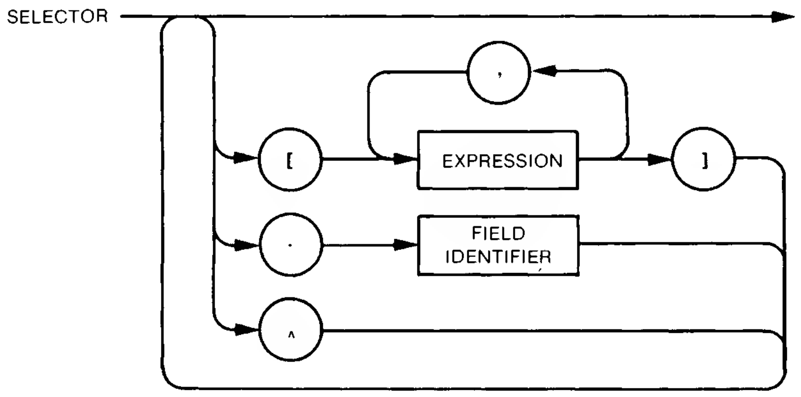


Figure 4-7. Selector Syntax

Array index computations are always performed with 16- or 32-bit index expressions. If a **BYTE** variable or expression is used as an index expression, the compiler generates run-time code to convert the byte value to a 16-bit value before computing the array element address.

Record Selector

A field of a record is selected by following the record identifier with a period (.) and the name of the field. The record name appearing before the period may itself be a selected variable. The **WITH** statement may be used to "open the scope" of the record, making it unnecessary to mention the record when accessing its fields.

Pointer Dereferencing

A pointer points to, or "references" a variable in the heap. To access that variable, the pointer is followed by the caret (^). No run-time checks are performed to make sure the pointer is not **NIL**. The pointer may itself be a selected variable.

Examples:

```
  p^  r.q^  ra[i].g^  ra[i].q^p^
```

File Buffer Selection

Every file in a program has implicitly associated with it a "buffer variable". This is the variable through which data is passed to or from a file. The file component at the current position of the file can be read into the buffer variable or the next item to be written to the file may be assigned to the variable and then written. The buffer variable, which is of the same type as the file base type, is denoted by following the file identifier with a caret (^). The file identifier appearing before the caret may itself be a selected variable.

Examples:

```
      fl^  f.ff^
```

Functional Type Change

A function type change causes an expression of one type to be treated as an expression of a different type. A functional type change consists of the type identifier of the result type, a left parenthesis, an expression, and a right parenthesis. Functional type changing is a Pascal/64000 extension of standard Pascal.

Examples:

```
$EXTENSIONS ON$
 VAR  U : UNSIGNED__8;
    I : INTEGER;
    A : ARRAY[0..9] OF INTEGER;
 BEGIN
    I := INTEGER (U);
    I := INTEGER (A);
    U := UNSIGNED__8(I + 4);
```

Functional type changing defeats the strong type checking inherent in the Pascal language and is therefore a potentially dangerous feature. Because there is a large number of combinations of input and output types, it is impossible to document all of the possible effects of functional type changing. There are two general cases (described below) which are handled differently by the compiler:

The first case occurs if both the source type and the result type are in the following list:

INTEGER, BYTE, SIGNED__8, SIGNED__16. SIGNED__32, UNSIGNED__8, UNSIGNED__16, UNSIGNED__32, CHAR, user-defined enumerated types, subranges of the preceding types, REAL, or LONGREAL.

In this case, the compiler may generate instructions to convert the source type to the result type. The intent of the conversion is to translate the numeric value of the source expression into the equivalent numeric value of the result type. For example, an ordinal expression may be converted to a real value.

If either the source type or the result type is not in the list above, then no conversion instructions are generated. The compiler simply treats the source expression as if it were an expression of the result type.

In either situation, the programmer should inspect the instructions generated by the compiler to insure that unexpected results fo not occur.

NOTES

Chapter 5

PROGRAMMING WITH PASCAL/64000

INTRODUCTION

The Pascal/64000 compiler will run on any HP 64000 system that has expanded host memory capability. The compiler accepts as input a sequence of statements from one or more source code files for conversion into a quasi-machine code which is stored in the 64000 for future use. This chapter discusses programming features that must be considered when writing a source program.

IDENTIFIERS

Identifiers are selected by the programmer to denote constants, types, variables, procedures, functions, and programs. An identifier consists of a letter followed by any combination of upper-case or lower-case letters, digits, or underscore (__). The syntactical construction of an identifier is shown in figure 5-1.

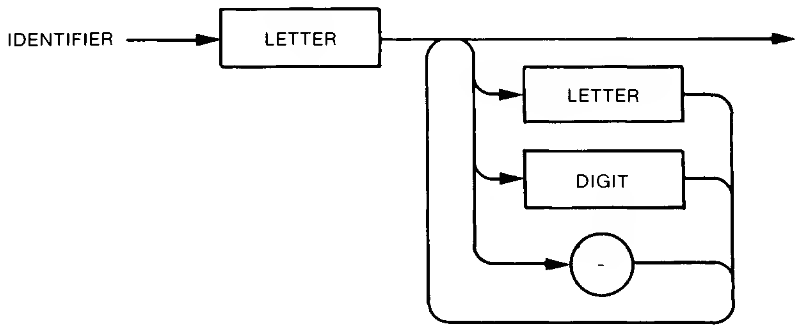


Figure 5-1. Identifier Syntax

When constructing identifiers, the following rules should be observed:

a. The first character of an identifier must be a letter.

b. Identifiers may contain any number of characters up to a source line in length; however only the first 15 characters are significant.

c. A reserved word cannot be used as an identifier.

d. Upper and lower case letters are differentiated within identifiers.

e. Each identifier must be unique within its scope (i.e., with a procedure or function in which they are defined).

f. All identifiers must be defined before they are used, except that a pointer type identifier may refer to a type that is defined later in the same declaration section.

PREDEFINED IDENTIFIERS

There are certain predefined identifiers that will be recognized by the Pascal/64000 compiler without being defined in the program. These predefined identifiers are listed in the following paragraphs.

Predefined Procedures

The following list of predefined procedures will be recognized by the compiler without further definition (refer to Chapter 6 for an explanation of each procedure):

APPEND	CLOSE	DISPOSE	GET
MARK	NEW	OPEN	OVERPRINT
PAGE	PROMPT	PUT	READ
READDIR	READLN	RELEASE	RESET
REWRITE	SEEK	STRREAD	STRWRITE
WRITE	WRITEDIR	WRITELN	

Predefined Functions

The following list of predefined functions will be recognized by the compiler without further definition (refer to Chapter 7 for an explanation of each function):

ABS	ADDR	ARCTAN	CHR	COS
EOF	EOLN	EXP	LINEPOS	LN
MAXPOS	ODD	ORD	POSITION	PRED
ROTATE	ROUND	SHIFT	SIN	SQRT
SUCC	TRUNC			

Predefined Files

The following predefined files will be recognized by the compiler without further definition.

INPUT	OUTPUT

Predefined Types

The following list of predefined types will be recognized by the compiler without further definition (refer to Chapter 1 for an explanation of each type):

BOOLEAN	BYTE	CHAR	INTEGER
LONGREAL	REAL	SIGNED_8	SIGNED_16
SIGNED_32	STRING	TEXT	UNSIGNED_8
UNSIGNED_16	UNSIGNED_32		

Predefined Constants

The following list of predefined constants will be recognized by the compiler without further definition (refer to Chapter 2 for an explanation of each constant):

FALSE	MAXINT	MININT	TRUE

Directives

A directive introduces a procedure or function declaration for which there is no block specified.

FORWARD	Indicates to the compiler that a block for the routine appears later in the program.
EXTERNAL	Indicates to the compiler that the routine is defined in some other module.

CHARACTER SET

Alphabetic Characters

The alphabetic characters include all upper and lower case characters ('A' thru 'Z' and 'a' thru 'z').

Numeric Characters

The numeric characters include the digits 0 thru 9 for decimal numbers, including A through F for hexadecimal numbers.

Special Characters

The special character (symbols) set and their use in Pascal programming are described as follows:

Character (Symbol)	Description	Use
'	Apostrophe	String Literal Delimiter.
*	Asterisk	Arithmetic operator - multiply; set intersection.
(*...*)	Asterisk pair	Comment delimiters.
{...}	Braces	Comment delimiters.
[...]	Brackets	Set constructor; array index delimiters.
:	Colon	Case constant list delimiter; statement label delimiter; field width delimiter; identifier list delimiter.
,	Comma	Argument list separator; enumerated type list separator.
$	Dollar sign	Compiler option (directive) delimiter.
=	Equal sign	Equality (relational operator).
-	Minus sign	Arithmetic operator - subtract/negate; set difference.
(...)	Parentheses	Delimits a parameter list or an expression group; delimits an enumerated type.
.	Period	End of program; decimal point; field selector.
+	Plus sign	Arithmetic operator - add; string concatenation; set union.
;	Semicolon	Parameter separator; statement separator.

/	Slant bar	Arithmetic operator - real divide.
_	Underscore	Allowed in identifiers but no as first character.
^	Caret	Indicates file buffer accessing; indicates pointer dereferencing.
:=	Symbol	Assignment indicator.
>	Symbol	Greater than (relational operator).
>=	Symbol	Greater than or equal to (relational operator); superset of.
<	Symbol	Less than (relational operator).
<=	Symbol	Less than or equal to (relational operator); subset of.
<>	Symbol	Not equal (relational operator).
..	Symbol	Subrange.

RESERVED WORDS

The following words are reserved. They have special meaning to the Pascal/64000 compiler and cannot be used as identifiers or user-defined symbols.

Reserved Word(s)	Description
AND	Boolean conjunction operator.
ARRAY	A structured type.
BEGIN, END	Delimit a compound statement.
CASE, OF, OTHERWISE	A conditional statement.
CONST	Indicates constant definition section.
DIV	Integer division operator.
FILE	A structured type.
FOR, TO, DOWNTO, DO	A repetitive statement.

FUNCTION	Indicates a function declaration.
GOTO	Control transfer statement.
IF, THEN, ELSE	A conditional statement.
IN	Set inclusion operator.
LABEL	Indicates label definition section.
MOD	Integer modulus operator.
NIL	Special pointer value.
NOT	Boolean negation operator.
OR	Boolean disjunction operator.
PROCEDURE	Indicates a procedure declaration.
PROGRAM	Program heading.
RECORD	A structured type.
REPEAT, UNTIL	A repetitive statement.
SET	A structured type.
TYPE	Indicates a type definition section.
VAR	Indicates a variable declaration section.
WHILE, DO	A repetitive statement.
WITH, DO	Opens record scope(s).

NUMBERS

PASCAL/64000 recognizes integers and real numbers. An integer is entered in binary, decimal, octal, or hexadecimal form. A real number can be specified as single or double percision. The paragraphs following describe the format and acceptable range of values for integers and real numbers.

Integers

Integers are positive and negative whole numbers ranging from -2^{31} to $2^{31} - 1$. This range contains numbers from -2,147,483,648 (MININT) through 2,147,483,647 (MAXINT). A minus sign (-) must precede a negative integer. A plus sign (+) may precede a positive integer, but it is not required. No commas or decimal points are allowed.

In addition to decimal integers, Pascal/64000 allows integers to be specified in binary, octal, and hexadecimal notation (see figure 5-2 for the integer syntax). To specify a binary integer, end the integer with the letter B. To specify an octal integer, end the integer with either the letter O or the letter Q. To specify a hexadecimal integer, end the integer with the letter H. When specifying an integer as binary, octal, or hexadecimal, the designated letter (B, H, O, or Q) must be upper-case. When no suffix is assigned, the decimal value is assumed.

NOTE

It is necessary to start a hexadecimal term with a decimal digit since the compiler will identify a term that starts with an alphabetic character as a label or an expression.

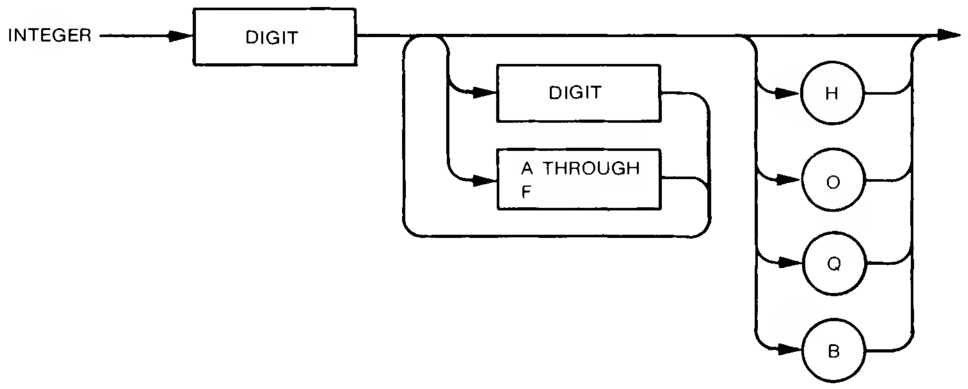


Figure 5-2. Integer Syntax

For microprocessors that do not allow 32-bit integers, the number 32768 (8000H) can only be interpreted as a negative value since its sign bit is set. The expression -32768 is a legal value, but it is scanned as being the negation of the positive value 32768. As a result, the compiler first detects it as the "out of range" positive value and gives the user an appropriate warning message:

"506: Warning: +32768 is treated as -32768 by the compiler"

NOTE

The warning is not printed if the microprocessor allows 32-bit integers.

As long as the user really wants the value -32768, the warning message may be ignored. The user will be able to suppress this message entirely if the **$EXTENSIONS$** option was enabled and 8000H was used to express the value -32768. The user could also express this constant as the constant expression <-MAXINT-1>, where **MAXINT** is a predefined constant with the value 32767 (7FFFH).

Real Numbers

In a Pascal program, real numbers are written in decimal or scientific notation. The following numbers are in decimal notation:

```
 5.1
 903.21345
-0.01
 6.0
```

Note that in decimal notation, at least one digit must appear on each side of the decimal point. For example, a zero must always precede the decimal point of a number between 1 and -1 and a zero must follow the decimal point of a whole number quantity (see figure 5-3 for real number syntax).

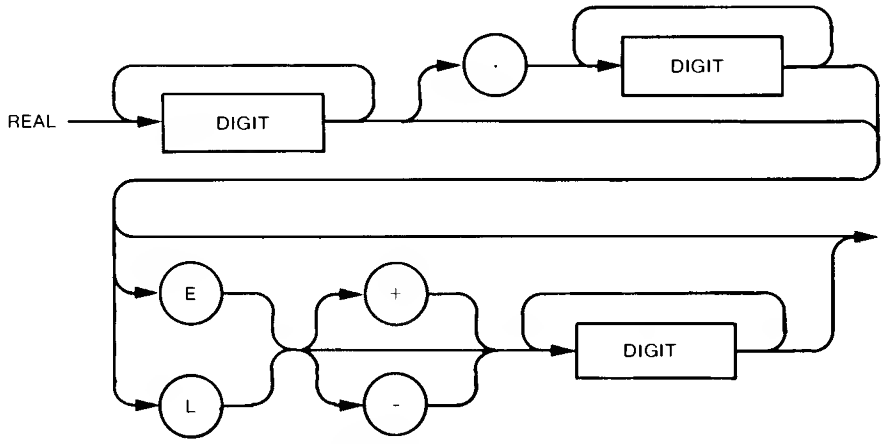


Figure 5-3. Real Syntax

Pascal/64000 also provides scientific (or exponential) notation as another way of writing real numbers. In scientific notation, the numbers are positive or negative values followed by an exponent. Use of the letter E after the value indicates the value is to be multiplied by a power of 10. The integer following the letter E indicates which power of 10 is positive or negative.

Examples:

```
1.2345E2
0.001122E6
3315E-1
```

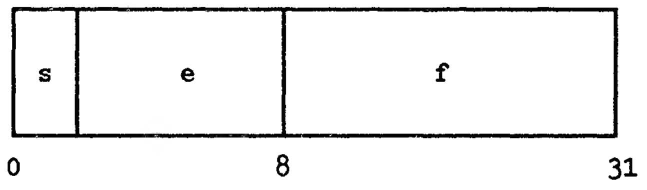
The scientific notation is referred to as a floating-point format because the position of the decimal point "floats" depending on the exponent following the letter E. At least one digit must appear on each side of the decimal point. In addition Pascal/64000 provides single and double precision for real numbers (refer to the paragraphs that follow for a detailed description of single and double precision real numbers). To indicate a double-precision real number, the user must use the floating-point notation, replacing the letter E with an uppercase letter L. The integer following the letter L is an exponent, as in single-precision (E) floating-point numbers.

Floating Point Numbers

IEEE format floating point numbers are supported for all processors. All arithmetic and relational operations are supported as well as the functions **SIN, COS, ARCTAN, EXP, LN, SQRT, ROUND,** and **TRUNC**.

Two packed formats (single precision and double precision) are supported as follows:

Single Precision Format - A single precision floating point number is a 32-bit binary value packed as shown below:

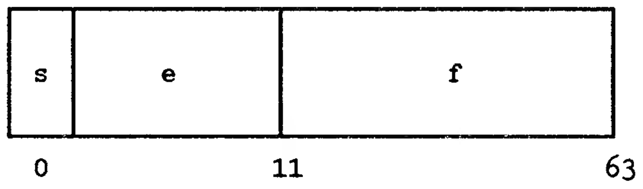


s is the sign bit.
e is the exponent.
f is the 23-bit fraction.

The value v of a single precision floating point number x can be computed as follows:

(a) If e=255 and f<>0, then v = not a number.
(b) If e = 255 and f = 0, then v = $(-1)^s$ +/- ∞.
(c) If 0 < e < 255, then v = $(-1)^s\,2^{e-127}$ (1.f).
(d) If e = 0 and f<>0, then v = $(-1)^s\,2^{-126}$ (0.f).
(e) If e = 0 and f = 0, then v = $(-1)^s$ 0 (zero).

Double Precision Format - A double precision floating point number is a 64-bit binary value packed as shown below:



s is the sign bit.
e is the exponent.
f is the 52-bit fraction.

The value v of a double precision floating point number x can be computed as follows:

(a) If e = 2047 and f<>0, then v = not a number.
(b) If e = 2047 and f = 0, then v = $(-1)^s$ +/- ∞.
(c) If 0 < e < 2047, then v = $(-1)^s\ 2^{e-1023}$ (1.f).
(d) If e = 0 and f<>0, then v = $(-1)^s\ 2^{-1022}$ (o.f).
(e) If e = 0 and f = 0, then v = $(-1)^s$ 0 (zero).

STRING LITERALS

A string literal is a sequence of ASCII characters enclosed by single or double quote marks (see figure 5-4 for String Literal syntax). String literals are constants of the string type. String literals containing a single character may also be of the predefined type **CHAR**.

a. If a string literal is to contain a single quote mark, it should be delimited by double quote marks and vice versa.

b. String literals must be contained on a single line.

Additional information on string literals and character constants may be found in Chapter 4.

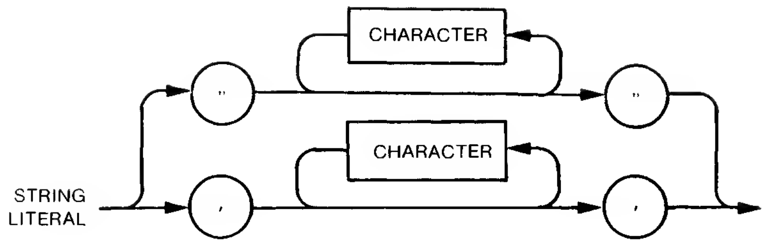


Figure 5-4. String Literal Syntax

COMMENTS

Words and messages contained in braces {...}, or parentheses/asterisks (*...*), are comments used to document the program. Comments are ignored by the compiler. A comment has the form:

{character string} or (*character string*)

Conventions to be observed when using comments are listed below:

a. Since a comment is equivalent to a blank, it may be placed anywhere in the program that a blank is permitted.

b. A comment beginning with a left brace ({) must terminate with a right brace (}). A comment beginning with the compound symbol (* must terminate with the compound symbol *).

SEPARATORS

A separator is a blank, an end-of-line marker, a comment, or a compiler option.

At least one separator must appear between any pair of consecutive identifiers, numbers, or reserved words. When one or both elements are special symbols, the separator is optional.

NOTES

Chapter 6

I/O

OVERVIEW

Pascal/64000 I/O characteristics are divided into two classes. The first class of behavior is that defined by the Pascal language as expressed in the HP Pascal Language Standard document. The second class is defined by the hardware and software I/O facilities available to the target microprocessor at run-time. The first class of behavior is independent of the target environment. It is implemented by the Pascal/64000 compiler and the Pascal I/O Library. The second class of behavior is dependent on the target environment. It is implemented for the 64000's emulation environment by the Simulated I/O Library. (See figure 6-1.)

Relationship of Pascal/64000 Program and Libraries.

There are two libraries that are used with Pascal/64000 programs to provide I/O capabilities. These are the Pascal I/O Library and the Simulated I/O Library. Library characteristics are listed below.

Pascal I/O Library Characteristics:

- Independent of target system I/O hardware or software interface.
- Called directly by Pascal/64000 program. For instance, calling RESET in the Pascal program causes the compiler to generate a call to "Preset" in the Pascal I/O library.
- Routines do not perform I/O directly but rather call "primitive" routines that exist in another library. For example, the routine Preset calls the primitive routine "open".

Simulated I/O Library Characteristics:

- Totally dependent on the 64000's emulation environment and the simulated I/O facilities that are a part of that environment.
- Not called directly by the Pascal/64000 but called by the Pascal I/O library routines. The Simulated I/O Library implements the "primitive" routines described above for the emulation environment.

  Performs I/O directly by manipulating the simulated I/O interface.

The Pascal/64000 I/O facilities were designed to achieve three primary objectives. These objectives are:

a. To provide Standard Pascal I/O constructs that operate uniformly for all target microprocessors and target system I/O implementations. A Pascal program using I/O should behave the same if it is compiled for the 6800, 68000, or any of the supported target processors. A Pascal program using I/O should behave the same when it is running in the emulation environment using simulated I/O or in the customer's target environment using the target system's I/O facilities.

b. To provide a quick and easy means of using the 64000's simulated I/O facilities in the emulation environment. A Pascal program should be able to access the 64000's keyboard, display, printer, rs232 port and disc files with a minimum of work. It should be unnecessary for the user to have any knowledge of the simulated I/O interface.

c. To provide an easy migration path from the emulation environment to the customer's target environment. When moving out of the emulation environment, the user should only have to rewrite the ten or so routines that are contained in the simulated I/O library. These "primitive" routines are the only ones that should have any dependency on the I/O environment.

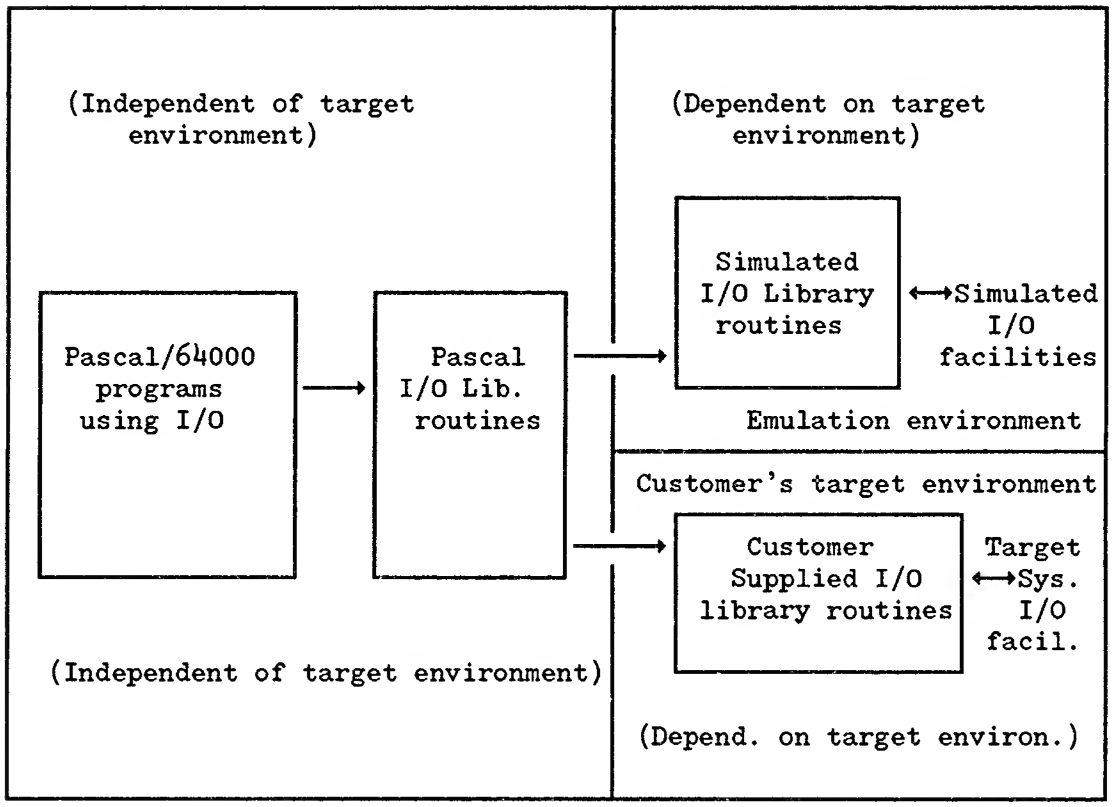

(Independent of target environment)
(Dependent on target environment)
Simulated I/O Library routines
Simulated I/O facilities
Pascal/64000 programs using I/O
Pascal I/O Lib. routines
Emulation environment
Customer's target environment
Customer Supplied I/O library routines
Target Sys. I/O facil.
(Independent of target environment)
(Depend. on target environ.)

Figure 6-1. Block Diagram of Pascal/64000 and I/O Libraries

The following example is written for the Motorola 68000 microprocessor. One can recompile it for any of the other microprocessors supported by the 64000.

```
"68000"

{**********************************************************************}
{                                                                      }
{   The following program can copy any 64000 file to any other 64000 }
{ file.                                                                }
{                                                                      }
{   COPY accepts the name of the source and destination files from    }
{ the 64000 keyboard and writes error and status messages to the      }
{ 64000 display.  The program terminates if it detects end-of-file    }
{ (i.e. an empty line) while reading either the source or destination}
{ file names.                                                          }
{                                                                      }
{   This program illustrates the use of variable length record mode. }
{ It is dependent on the Simulated I/O environment for two reasons:   }
{     a. Simulated I/O defines physical records.                      }
{     b. Simulated I/O physical records have a maximum length of      }
{        256 bytes.                                                    }
{                                                                      }
{**********************************************************************}

PROGRAM COPY;  (NOTE: Program parameters are not implemented in       }
               {      Pascal/64000 because program parameters imply  }
               {      the existence of an "operating system" which   }
               {      may not exist in the target environment.  One  }
               {      may use the predefined variables INPUT and     }
               {      OUTPUT without mentioning them in the program  }
               {      parameter list.                                 }
TYPE
  REC_TYPE      = ARRAY[0..255] OF CHAR; (Simulated I/O record         }
  FILE_TYPE     = FILE OF REC_TYPE;      (Non-text file                }

{ The following types are needed for the Simulated I/O interface.     }
{ The CA_BUFFER is defined here to be 258 bytes long.  CA buffers     }
{ for some devices may actually be shorter.  The display requires     }
{ 257 bytes, the printer requires 242 bytes, the rs232 requires 240   }
{ bytes, the keyboard 243 bytes, and only disc files require 258.     }

  CA_PTR        = ^CA_BUFFER;            {Pointer to CA_BUFFER.        }
  CA_BUFFER     = ARRAY[0..257] OF CHAR; {Simulated I/O CA buffer      }

VAR
  BUFFER              : REC_TYPE;    {Buffer for data to be copied     }
  DONE                : BOOLEAN;     {TRUE if end-of-file detected     }
  SOURCE_OPEN         : BOOLEAN;     {TRUE if source file opened       }
  DESTINATION_OPEN    : BOOLEAN;     {TRUE if destination file opened  }
  SOURCE              : FILE_TYPE;   {Source file variable             }
  DESTINATION         : FILE_TYPE;   {Destination file variable        }
  SOURCE_NAME         : STRING;      {Source file name                 }
  DESTINATION_NAME    : STRING;      {Destination file name            }
```

```
$EXTVAR ON$  {Following variables are defined in the Pascal I/O Lib. }
  READ_REC_LEN        : SIGNED_16;  {Contains no. of bytes transferred}
                                    {to buffer variable during last   }
                                    {input operation.                 }
  WRITE_REC_LEN       : SIGNED_16;  {Set to contain the no. of bytes  }
                                    {to be written in next 'VARIABLE' }
                                    {mode output operation.           }
$EXTVAR OFF$

{   The following variables are the CA buffers needed by Simulated    }
{ I/O.  They are ORGed at specific addresses because their addresses }
{ need to be known during emulation configuration and it is          }
{ convenient if the CA buffers do not move if the program is changed.}

$ORG 7000H$
  DISPLAY_CA          : CA_BUFFER;  {CA buffer for display            }
$ORG 7102H$
  PRINTER_CA          : CA_BUFFER;  {CA buffer for printer            }
$ORG 7204H$
  RS232_CA            : CA_BUFFER;  {CA buffer for rs232              }
$ORG 7306H$
  KEYBOARD_CA         : CA_BUFFER;  {CA buffer for keyboard           }
$ORG 7408H$
  DISC_1_CA           : CA_BUFFER;  {CA buffer for disc file #1       }
$ORG 750AH$
  DISC_2_CA           : CA_BUFFER;  {CA buffer for disc file #2       }
$END_ORG$
PROCEDURE Pinit_pascal_io;      EXTERNAL; {Pascal I/O Library routine }
                                          {to initialize its data     }
                                          {structures.                }
PROCEDURE Pshutdown_pascal_io; EXTERNAL; {Pascal I/O Library routine }
                                          {to close all open file     }
                                          {variables.                 }
FUNCTION  IORESULT:BYTE;        EXTERNAL; {Pascal I/O Library routine }
                                          {returns result code for    }
                                          {most recent I/O operation. }

{   Simulated I/O Library routine to initialize its data structures. }
{Its parameters are the addresses of the CA buffers required for the }
{simulated I/O interface.  NIL indicates that no CA is allocated.    }

PROCEDURE INIT_SIMIO_LIB(DISPLAY_CA,  {display      CA buffer address}
                         PRINTER_CA,  {printer      CA buffer address}
                         RS232_CA,    {rs232        CA buffer address}
                         KEYBOARD_CA, {keyboard     CA buffer address}
                         DISC_1_CA,   {disc file #1 CA buffer address}
                         DISC_2_CA,   {disc file #2 CA buffer address}
                         DISC_3_CA,   {disc file #3 CA buffer address}
                         DISC_4_CA,   {disc file #4 CA buffer address}
                         DISC_5_CA,   {disc file #5 CA buffer address}
                         DISC_6_CA    {disc file #6 CA buffer address}
          : CA_PTR); EXTERNAL;
```

```
BEGIN {COPY}

Pinit_pascal_io;          {Initialize Pascal I/O Library data.      }

INIT_SIMIO_LIB(           {Initialize Simulated I/O Library data.   }
               ADDR(DISPLAY_CA),        {Ptr to display      CA buff }
               ADDR(PRINTER_CA),        {Ptr to printer      CA buff }
               ADDR(RS232_CA),          {Ptr to rs232        CA buff }
               ADDR(KEYBOARD_CA),       {Ptr to keyboard     CA buff }
               ADDR(DISC_1_CA),         {Ptr to disc file #1 CA buff }
               ADDR(DISC_2_CA),         {Ptr to disc file #2 CA buff }
               NIL,                     {disc file # 3 not needed    }
               NIL,                     {disc file # 4 not needed    }
               NIL,                     {disc file # 5 not needed    }
               NIL);                    {disc file # 6 not needed    }

RESET(INPUT,'keyboard');    {NOTE: One must explicitly open INPUT.    }
REWRITE(OUTPUT,'display'); {NOTE: One must explicitly open OUTPUT.    }
DONE := FALSE;

WHILE NOT DONE DO
  BEGIN {NOT DONE}

  {Accept and open source file}
  SOURCE_OPEN := FALSE;
  WHILE NOT (DONE OR SOURCE_OPEN) DO
    BEGIN {OPEN SOURCE FILE}
    WRITELN('Type in source file name:(Empty line terminates program)');
    IF EOF THEN                         {Check EOF before reading.    }
      DONE := TRUE
    ELSE
      BEGIN {NOT EOF}
      READLN(SOURCE_NAME);              {Read source file name.       }
      $IOCHECK OFF$                     {Turn off I/O error checking }
      RESET(SOURCE,SOURCE_NAME,'VARIABLE');{Attempt to open source file.}
      $IOCHECK ON$                      {Turn on I/O error checking. }
      IF IORESULT = 0 THEN
        BEGIN {RESET SUCCESSFUL}
        SOURCE_OPEN := TRUE;
        WRITELN('File "',SOURCE_NAME,'" opened for input.');
        END   {RESET SUCCESSFUL}
      ELSE
        WRITELN('Unable to open file "',SOURCE_NAME,'" for input.');
      END;  {NOT EOF}
    END;  {OPEN SOURCE FILE}

  {Accept and open destination file}
  DESTINATION_OPEN := FALSE;
  WHILE NOT (DONE OR DESTINATION_OPEN) DO
    BEGIN {OPEN DESTINATION FILE}
    WRITELN('Type in destination file name:(Empty line terminates program)');
    IF EOF THEN                         {Check EOF before reading.    }
      DONE := TRUE
    ELSE
      BEGIN {NOT EOF}
      READLN(DESTINATION_NAME);          {Read destination file name. }
      $IOCHECK OFF$                      {Turn off I/O error checking }
      REWRITE(DESTINATION,DESTINATION_NAME,'VARIABLE');{Attempt to    }
                              {open destination file.               }
      $IOCHECK ON$                       {Turn on I/O error checking. }
      IF IORESULT = 0 THEN
        BEGIN {RESET SUCCESSFUL}
        DESTINATION_OPEN := TRUE;
        WRITELN('File "',DESTINATION_NAME,'" opened for output.');
        END   {RESET SUCCESSFUL}
      ELSE
        WRITELN('Unable to open file "',DESTINATION_NAME,'" for output.');
      END;  {NOT EOF}
    END;  {OPEN DESTINATION FILE}

  {Copy source file to destination file}
  IF SOURCE_OPEN AND DESTINATION_OPEN THEN
    BEGIN {COPY FILES}
    WHILE NOT EOF(SOURCE) DO
      BEGIN {NOT EOF(SOURCE)}
      READ(SOURCE,BUFFER);           {Read record into buffer          }
      WRITE_REC_LEN := READ_REC_LEN;{Output record same size as input}
      WRITE(DESTINATION,BUFFER);     {Write record from buffer         }
      END;  {NOT EOF(SOURCE)}
    CLOSE(SOURCE);
    CLOSE(DESTINATION);
    WRITELN('"',SOURCE_NAME,'" copied to "',DESTINATION_NAME,'"');
    END;  {COPY FILES}
  END;  {NOT DONE}

Pshutdown_pascal_io;       {Close all file variables.                 }

END.  {COPY}
```

PASCAL/64000 I/O.

The following section defines terms and introduces some concepts that will be used throughout the I/O chapter.

Logical Files and Physical Files

The word "file" has several meanings in the context of a Pascal program. Therefore, definitions of the terms "logical file" and "physical file" are in order so that the following discussion will be more precise.

A logical file is a variable that exists within a Pascal/64000 program. The characteristics of a logical file are defined by the Pascal Language and are independent of any Pascal environment. Logical files are referenced by identifiers just like any other Pascal variable. A synonym for logical file is "file variable." An example of a logical file definition is the following.

```
TYPE T = INTEGER;   {Can be any type not containing a FILE};
VAR  F : FILE OF T;   {F is a logical file}
```

A physical file exists independently of Pascal or any other program. Its characteristics are defined by a particular computer system and are independent of any program or language which accesses it. A physical file is identified by a "file name" which is a string of characters that can uniquely identify a file within a particular environment. Examples of file names on the 64000 system are the following.

```
COPY:IODOC:0:source
printer
```

At run-time, a logical file is "associated" with a physical file using the opening procedures RESET, REWRITE, APPEND, or OPEN. When a logical file is open, it can read and write data to a physical file. This association lasts until the file is closed.

Logical File States

A logical file (i.e. a file variable) always has one of four states in a Pascal program: closed, read-only, write-only, or read-write.

closed	- A closed file variable is not associated with any physical file. An error will result if any input or output operation is executed on that file.
read-only	- A read-only file variable is open (i.e. associated with a physical file). Only input operations are allowed. The procedure RESET opens a file in the read-only state.
write-only	- A write-only file variable is open (i.e. associated with a physical file). Only output operations are allowed. The procedures REWRITE or APPEND open a file in the write-only state.
read-write	- A read-write file variable is open (i.e. associate with a physical file). Both input and output operations are allowed. In addition, the routines SEEK, READDIR, WRITEDIR, and MAXPOS may be performed on read-write file. The procedure OPEN opens a file in the read-write state. Note that textfiles (defined later) may not be opened in the read-write state.

Logical Records, Physical Records, and Variable Length Records

A logical file consists of a sequence of components all of the same type. The components are numbered beginning with component number 1. Sometimes the term "logical record" is used as a synonym for file component. Only one component of a file, the current component, is accessible at a time. Given a variable "F: FILE OF T;" where T is some type, then all the components of the file are of type T. The length of each component in bytes is the length of T. If T is a record with variants, then the length of T is the length of the largest variant of T.

A particular I/O implementation may or may not divide the data in physical files into "physical records". Some file systems do not define physical records at all but consider a file to be just a sequence of data bytes. The 64000 file system used by the simulated I/O library, defines physical records. For 64000 disc files, a physical file consists of a number of variable length physical records, each record containing from 2 to 256 bytes of data.

The Pascal language does not recognize physical records. It treats files as a stream of data bytes. All logical records have a fixed length defined by the Pascal program. When reading a logical record, Pascal reads as many physical records as necessary to fill the logical record. When writing a logical record, Pascal writes physical records as necessary to output the data. In Pascal, therefore, the logical record definition in the program overrides any implementation of physical records in the I/O system.

Sometimes it is important to the programmer to understand and deal with physical records. This is particularly true when physical files have been produced by other language systems or when a physical file is written using one Pascal file variable and read using a variable with a different length component. Pascal/64000 has non-standard features which allow the programmer to deal with physical records.

A logical file may be opened in "variable length record mode" using the procedures RESET, REWRITE, APPEND, or OPEN. For example,

```
        RESET(F,'FILENAME','VARIABLE');
```

In variable length record mode, the physical record definition, if it exists, overrides the logical record definition. When reading a component, data bytes are input until the logical record is full or until the end of a physical record is encountered, whichever comes first. When writing a component, the programmer specifies the length of the component and writes a physical record of that length.

The details of reading and writing files in variable length record mode are discussed later in this chapter.

The Buffer Variable

The "buffer variable" is a part of every file variable. Given the following definition of a file variable:

```
    VAR F: FILE OF T;
```

then the buffer variable is an ordinary variable of type T and is referenced by way of the construction F^. When reading a file, each file component is read into the buffer variable. When writing, the data for each component is assigned to the buffer variable before being written.

In Pascal/64000, because of the "deferred GET" implementation described later, a reference to the buffer variable F^ may cause an input operation to occur.

Textfiles and Non-text Files

The type identifier TEXT is predefined by the Pascal/64000 compiler as follows:

```
    TYPE TEXT = FILE OF CHAR;
```

File variables of type TEXT are called "textfiles". All other files are called "non-text files". Note the following definitions.

```
    VAR F: TEXT;
        G: FILE OF CHAR;
```

F is a textfile. G is a non-text file whose components happen to be of type CHAR. Only those variables defined in terms of the identifier TEXT are textfiles.

Textfiles have special properties. The components of a textfile, of type CHAR, are further structured into "lines" separated by "line markers". Line markers are written to a textfile using the predefined procedure WRITELN. Line markers are detected (when reading) by using the predefined function EOLN. If, when reading, the textfile F is positioned at a line marker, then the buffer variable F^ will contain the ASCII space character.

The actual implementation of line markers is not defined by the Pascal language. Various I/O environments may implement line markers differently. The Pascal I/O Library, discussed later in this chapter, contains provisions for dealing with different implementations.

Although the components of textfiles are of type CHAR, one may use the procedures READ and WRITE to transfer values of other types, INTEGER, UNSIGNED__xx, REAL, LONGREAL, and STRING, to and from textfiles. In this case, Pascal performs an implicit data conversion. When reading, a sequence of characters is converted to the appropriate internal representation for, for example, an INTEGER. When writing, the value of a REAL expression, for example, is converted to a sequence of characters.

"Deferred GET" Implementation of I/O

Jensen and Wirth in the "Pascal User Manual" define the procedure GET(F) as follows:

> GET(F) advances the file window to the next component; i.e. assigns the value of this component to the buffer variable F^.

This definition has unfortunate consequences for interactive devices such as terminal keyboards. For example, the procedure call READ(F,V) can be rewritten as follows:

```
V := F^;
GET(F);
```

Using the above definition for GET, in order to read the "present" character from the keyboard, one must additionally type in the "next" character before READ will return to the calling program. Coding around this problem leads to unnatural looking programs.

HP Standard Pascal addresses this problem with a "deferred GET" implementation. The following "deferred GET" definition postpones the input operation associated with the GET until the data from the physical file is actually needed.

> GET(F) cause a subsequent reference to the buffer variable or subsequent calls to the functions EOF(F) or EOLN(F) to actually move the current file component into the buffer variable and advance to the next component. If the current component does not exist, F^ will be undefined and EOF(F) will return TRUE. An error will occur if F is not open for reading or if EOF(F) was TRUE prior to the call.

This definition results in the following sequence of events when reading a file:

a. The program executes a call to GET. A flag is set within the file variable to remember this event. No input operation is performed.

b. The program executes one of the following operations: a reference to the buffer variable, a call to EOF, or a call to EOLN. Since the GET flag is set, an input operation is performed which advances the file position and transfers this file component to the buffer variable. The GET flag is reset.

c. Subsequent references to the buffer variable or calls to EOF or EOLN do not perform input operations since the GET flag is reset and the present component is already in the buffer variable. In order to advance to the next component, it is necessary to call GET again.

I/O Error Handling

The Pascal Language does not define any features for dealing with errors of any kind including I/O errors. A practical language system must, however, be able to deal with I/O errors within the program. The compiler option $IOCHECK$ allows the programmer to select between two methods of handling I/O errors. One may mix these two methods within the same program as needed by setting IOCHECK ON or OFF.

With $IOCHECK ON$, the default case, if a Pascal I/O library routine detects an error, the program terminates. The exact method of termination depends on the particular target microprocessor and its environment. For example, for the Motorola 68000 in the 64000 emulation environment, the error handling routines builds an error message, "I/O error NN at XXXXXXXX" and calls a routine in the emulation monitor. Other processors use different methods, but, in general, I/O errors will be handled just like any other run-time library error for that processor.

If a Pascal I/O library routine detects an error with $IOCHECK OFF$, the routine sets a global variable to a result code and returns. The programmer should then call the function IORESULT, defined in the Pascal I/O library, to obtain the result code for the most recent I/O operation. For example,

```
VAR F,G: TEXT;
FUNCTION IORESULT:SIGNED__8; EXTERNAL;
BEGIN
RESET(F,'NAME1');     {Error detected in this call terminates program}
$IOCHECK OFF$         {Turn off IOCHECK option}
RESET(G,'NAME2');     {This call just sets a result code}
IF IORESULT <> 0 THEN {Check result of last I/O operation}
  BEGIN {Handle I/O error} END;
```

The value returned by IORESULT has the following meanings:

0	-	Operation successful, no errors detected.
1	-	Error in I/O primitive routine. A primitive routine, called by a Pascal I/O library routine detected an error. The nature of this error is defined by the I/O environment and the library routines that manipulate that environment. For example, the Simulated I/O library defines a global variable "errno" which contains the result code for the last simulated I/O operation.
2-n	-	Pascal I/O Library error. These errors are defined by the Pascal language for all I/O environments. An example is IORESULT = 2, an I/O operation was performed on a closed file.

Deviations From Standard Pascal

The following features of Pascal/64000 are different from those for HP Standard Pascal.

PROGRAM PARAMETERS NOT IMPLEMENTED. Pascal/64000 does not allow a list of program parameters to be specified after the program name. For example:

```
PROGRAM P(INPUT,OUTPUT); {Produces error number 450}
```

The existence of program parameters implies that there is an "operating system" which executes the Pascal program and passes values to the Pascal program. In a microprocessor environment, however, such an operating system may not exist. Therefore Pascal/64000 disallows program parameters. The predefined file variables INPUT and OUTPUT may still be used without being declared in the program and without being mentioned in the program parameter list.

STANDARD FILES INPUT AND OUTPUT MUST BE OPENED EXPLICITLY. The Pascal/64000 compiler does not automatically generate calls to RESET(INPUT) and REWRITE(OUTPUT) at the beginning of the main program block. The programmer must explicitly open the files INPUT and OUTPUT before using them.

PROGRAM LEVEL FILES AND DYNAMIC FILES MUST BE CLOSED EXPLICITLY. Pascal/64000 does not automatically close program level file variables upon leaving the body of the main program. The programmer must write explicit code to accomplish this. This is because Pascal/64000 allows separately compiled program modules to be linked together to form a complete program. The Pascal/64000 compiler, when compiling the main module, does not have knowledge of program level file variables in other program modules.

There are two methods of closing these files. The programmer may write explicit calls to CLOSE for each program level file variable. Also, the Pascal I/O Library defines the procedure Pshutdown__pascal__io which closes all files.

Pascal/64000 does not automatically close dynamically allocated file variables when the dynamic variable is disposed. If one allocates a file variable with the procedure NEW, then one must explicitly call CLOSE for that file variable before calling DISPOSE.

The Pascal/64000 compiler automatically generates code to close files that are local to a procedure or function at the end of the procedure or function body.

INPUT AND OUTPUT OF ENUMERATED TYPES TO TEXTFILES NOT IMPLEMENTED. HP Standard Pascal allows enumerated types to be used as parameters to READ and WRITE to textfiles. Pascal/64000 does not implement this feature.

Implementation Dependent Features

Although Pascal/64000 attempts to implement I/O in an implementation independent manner, this objective has not been achieved completely. The following features of the Pascal I/O library are dependent on the I/O environment.

a. Maximum number of file variables open at one time.
b. Maximum number of logical records in a non-text file.
c. Maximum number of characters in a textfile line.
d. Definition of a line marker for textfiles.
e. Padding of odd-length logical records.
f. Variable length records; determining and specifying their length.

In some cases, the behavior of implementation dependent features is controlled by global variables defined in the Pascal I/O Library. The globals are set to their default values by the routine Pinit__pascal__io in the Pascal I/O Library. The default values are selected to be compatible with the Simulated I/O environment. The programmer may, after calling Pinit__pascal__io, set the globals to different values to select different behavior. Refer to the Pascal I/O Libraries section in this chapter for more information.

Standard Procedures and Functions for I/O

APPEND

Usage:

```
APPEND(F)
APPEND(F,S)
APPEND(F,S,T)
```

Parameters:

F must be a file variable.

S must be a string expression. The contents of S should be the name of a physical file that will be associated with F. The rules for physical file names and their meanings are not specified by Pascal. The interpretation of physical file names is done in the I/O environment library. Only the first 49 characters of S are significant.

T must be a string expression. The contents of T specifies I/O implementation dependent information about the file F. The Pascal I/O library recognizes one value for T that specifies variable length record mode as opposed to fixed length records.

'VARIABLE' - specifies that physical records in the physical file will override the logical record description of the file variable. (Refer to the previous discussion on Logical/Physical/Variable Length Records.)

Other values for T will be ignored. The Pascal I/O library ignores leading and trailing blanks in T and considers upper and lower case letters equivalent.

Description:

The procedure APPEND(F) opens the file F in the write-only state and places the current file position immediately after the last existing component. If the physical file associated with F does not exist, then the physical file is created (if possible). EOF(F) now returns TRUE and the contents of the buffer variable F^ is undefined.

If the file F is already open when APPEND is called, then F is first closed.

If the parameter S is specified and it is not the null string or all spaces, then the content of S is used as the physical file name to be associated with F. If S is not specified or if S is the null string or all spaces, then the name of the physical file previously associated with F, if F was open, is used. If F was not open and S is not specified, then the null string (i.e. '') is used as the physical file name.

Simulated I/O Note: The null string is not a legal file name for simulated I/O and causes an error.

APPEND is an HP Standard Pascal extension of ANSI Pascal.

CLOSE

Usage:

```
CLOSE(F)
CLOSE(F,S)
```

Parameters:

F	must be a file variable.
S	must be a string expression. The contents of S specify the disposition of any physical file associated with F. The Pascal I/O Library recognizes one value.

PURGE - specifies that the physical file associated with F will be deleted from the file system.

Other values for S will be ignored. The Pascal I/O library ignores leading and trailing blanks In S and considers upper and lower case letters equivalent.

Description:

The procedure CLOSE(F) closes the file F and breaks any association between F and a physical file. After CLOSE, EOF(F) returns TRUE and the contents of the buffer variable F^ is undefined. If F was closed when CLOSE was called, no action is taken. If F was open and the contents of S was 'PURGE', then APPEND attempts to delete from the file system the physical file associated with F.

CLOSE is an HP Standard Pascal extension of ANSI Pascal.

EOF

Usage:

EOF(F)
EOF

Parameters:

F	must be a file variable. If F is omitted, the predefined variable INPUT is used.

Description:

The BOOLEAN function EOF(F) returns TRUE if the file F is closed or open in the write-only state. EOF(F) returns TRUE if F is open in the read-only or read-write states and the current file position is past the last existing component in the physical file associated with F. Otherwise, EOF(F) returns FALSE.

EOF references to the buffer variable F^ during its operation could possibly cause an input operation to occur.

EOLN

Usage:

EOLN(F)
EOLN

Parameters:

F	must be a textfile variable open in the read-only state. If F is omitted, the predefined variable INPUT is used.

Description:

The BOOLEAN function EOLN(F) returns TRUE if the current position of textfile F is at a line marker. Otherwise, EOLN(F) returns FALSE.

EOLN references to the buffer variable F^ during its operation could possibly cause an input operation to occur.

GET

Usage:
GET(F)

Parameters:

F	must be a file variable open in the read-only or read-write states.

Description:

The procedure GET(F) causes a subsequent reference to the buffer variable F^ or subsequent calls to the functions EOF(F) or EOLN(F) to perform an input operation which moves the current component into the buffer variable and advances to the next component. If the current component does not exist, the buffer variable F^ will be undefined and EOF(F) will return TRUE. An error will occur if F is not in the read-only or read-write states or if EOF(F) was TRUE prior to the call.

LINEPOS

Usage:
LINEPOS(F)

Parameters:

F	must be a textfile variable open in the read-only or write-only state.

Description:

The SIGNED__16 function LINEPOS(F) returns the number of characters read from or written to the textfile F since the last line marker. This does not include the character in the buffer variable F^. LINEPOS returns zero after reading a line marker or after a call to READLN or WRITELN.

LINEPOS is an HP Standard Pascal extension of ANSI Pascal.

MAXPOS

Usage:

    MAXPOS(F)

Parameters:

F	must be a non-text file variable open in the read-write mode.

Description:

The SIGNED__16 function MAXPOS(F) returns the number of the last component of F which may ever be accessed. Note that is not the number of components actually written in the physical file associated with F. It is the maximum number that may ever be written.

The value returned by MAXPOS is implementation dependent. Refer to the decription of the routine Pmaxpos presented later in this chapter for further details.

MAXPOS is an HP Standard Pascal extension of ANSI Pascal.

OPEN

Usage:

    OPEN(F)
    OPEN(F,S)
    OPEN(F,S,T)

Parameters:

F	must be a non-text file variable.
S	must be a string expression. The contents of S should be the name of a physical file that will be associated with F. The rules for physical file names and their meanings are not specified by Pascal. The interpretation of physical file names is done in the I/O environment library. Only the first 49 characters of S are significant.
T	must be a string expression. The contents of T specifies I/O implementation dependent information about the file F. The Pascal I/O library recognizes one value for T that specifies variable length record mode as opposed to fixed length records. 'VARIABLE' - specifies that physical records in the physical file will override the logical record description of the file variable (refer to description given previously).

Other values for T will be ignored. The Pascal I/O library ignores leading and trailing blanks in T and considers upper and lower case letters equivalent.

Description:

The procedure OPEN(F) opens the file F in the read-write state and places the current file position at the first component of the file. If the physical file associated with F does not exist, then the physical file is created (if possible). EOF(F) now returns FALSE, unless F is empty, and the contents of the buffer variable F^ is undefined.

If the file F is already open when OPEN is called, then F is first closed.

If the parameter S is specified and it is not the null string or all spaces, then the content of S is used as the physical file name to be associated with F. If S is not specified or if S is the null string or all spaces, then the name of the physical file previously associated with F, if F was open, is used. If F was not open and S is not specified, then the null string (i.e. '') is used as the physical file name.

Simulated I/O Note:	The null string is not a legal file name for simulated I/O and causes an error.
	The Simulated I/O Library does not fully implement read-write (i.e. random access) files. Refer to the Simulated I/O Libraries section in this chapter for important restrictions regarding positioning and writing on read-write files.

OPEN is an HP Standard Pascal extension of ANSI Pascal.

OVERPRINT

Usage:

```
OVERPRINT(F)
OVERPRINT
```

Parameters:

F	must be a textfile variable open in the write-only mode. If F is omitted, the predefined variable OUTPUT is used.

Description:

The procedure OVERPRINT(F) writes a special line marker to the textfile F. When the physical file is printed, this line marker causes the next line to be printed on top of the line just printed.

Simulated I/O Note:	The Simulated I/O Library does not implement OVERPRINT as described above. With simulated I/O, OVERPRINT is the equivalent of WRITELN.

OVERPRINT is an HP Standard Pascal extension of ANSI Pascal.

PAGE

Usage:

```
PAGE(F)
PAGE
```

Parameters:

F	must be a textfile variable open in the write-only mode. If F is omitted, the predefined variable OUTPUT is used.

Description:

The procedure PAGE(F) writes a special line marker to the textfile F. When the physical file is printed, this line marker causes the printer to skip to the top of the form.

POSITION

Usage:

```
POSITION(F)
```

Parameters:

F	must be a non-text file variable which is not closed.

Description:

The SIGNED__16 function POSITION(F) will be the current file position (i.e. component number) of F. The first component of a file is number 1.

POSITION is an HP Standard Pascal extension of ANSI Pascal.

PROMPT

Usage:

```
PROMPT(F)
PROMPT
```

Parameters:

F	must be a textfile variable open in the write-only mode. If F is omitted, the predefined variable OUTPUT is used.

Description:

The procedure PROMPT(F) causes the system to write any buffered data for textfile F to the physical file. A line marker is not written and the file position is not advanced.

Simulated I/O Note: The Simulated I/O Library does not implement PROMPT as described above. With simulated I/O, PROMPT is the equivalent of WRITELN.

PROMPT is an HP Standard Pascal extension of ANSI Pascal.

PUT

Usage:

```
PUT(F)
```

Parameters:

F	must be a file variable open in the write-only or read-write state.

Description:

The procedure PUT(F) writes the contents of the buffer variable F^ to the current component of the file and advances the file position to the next component.

Simulated I/O Note: The Simulated I/O Library does not fully implement read-write (i.e. random access) files. Refer to the section on Simulated I/O Library for important restrictions regarding positioning and writing on read-write files.

READ

Usage:

```
READ(F,V)
READ(F,V1, ... ,Vn)
READ(V)
READ(V1, ... ,Vn)
```

Parameters:

F	must be a file variable open in the read-only or read-write state. If the first parameter to READ is not a file, then the predefined variable INPUT is used.
V	must be a variable. The type of V depends on whether F is a textfile. If F is a non-text file, the component type of F must be compatible with the type of V. If F is a textfile, the type of V must be compatible with CHAR, INTEGER, UNSIGNED__nn, REAL, LONGREAL, or STRING. V may be a component of a packed structure. Any number of suitable variables may be specified.

Description:

The procedure READ(F,V) assigns the value of the current component of F to the variable V and advances to the next component. A subsequent reference to the buffer variable F^ will cause the next component to load the new current component into the buffer variable.

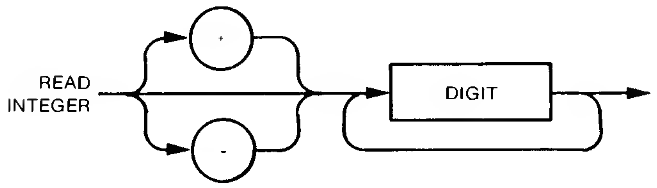
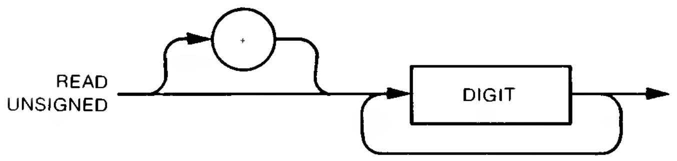
```
READ(F,V1,V2, ... ,Vn) is equivalent to  READ(F,V1);
                                         READ(F,V2);
                                         ...
                                         READ(F,Vn);

READ(V)                is equivalent to  READ(INPUT,V)

READ(V1,V2, ... ,Vn)   is equivalent to  READ(INPUT,V1);
                                         READ(INPUT,V2);
                                         ...
                                         READ(INPUT,Vn);
```

If F is a non-text file, the following equivalence describes the operation of READ.

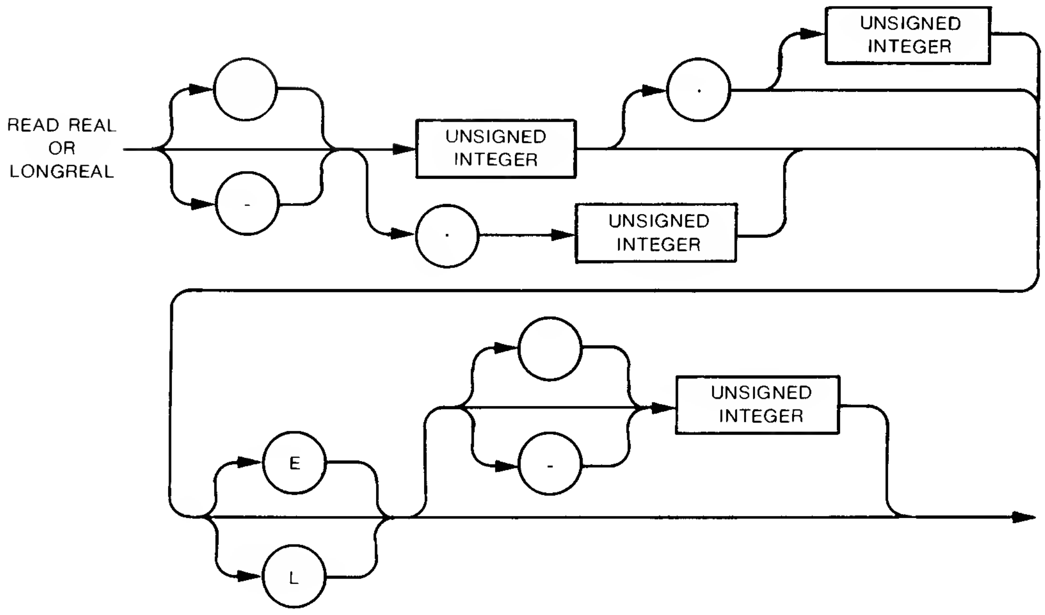
```
READ(F,V)           is equivalent to  V := F^;
                                      GET(F);
```

If F is a textfile, its components are of type CHAR. The variable V, however, may be of a type compatible with CHAR, INTEGER, UNSIGNED__nn, REAL, LONGREAL, or STRING. An implicit data conversion is performed from the characters of the textfile to the internal representation of the variable V. The rules of the conversion depend on the type of V.

After completion, F^ will contain the character following the last character used by READ. An error will occur if EOF(F) is TRUE before READ is called or if EOF(F) becomes true before finishing the read operation.

Type of V	Conversion Rules
CHAR subrange of CHAR	The current component of F (of type CHAR) is assigned to V and the file position is advanced. No conversion is performed.
INTEGER, BYTE, SIGNED__8, SIGNED__16, SIGNED__32, or a subrange of these types	READ skips preceding blanks and line markers and reads a sequence of characters described by the diagram below. The signed decimal value of these characters is stored in the variable V. An error occurs if the sequence described below is not satisfied. An error occurs if the value found is outside the range of INTEGER. 
UNSIGNED__8, UNSIGNED__16, UNSIGNED__32, or subrange of these types	READ skips preceding blanks and line markers and reads a sequence of characters described by the diagram below. The unsigned decimal value of these characters is stored in the variable V. An error occurs if the sequence described below is not satisfied. An error occurs if the value found is outside the range of UNSIGNED__32 (or UNSIGNED__16 for the 6800 and 6809). 

Type of V	**Conversion Rules**
REAL or LONGREAL	READ skips preceding blanks and line markers and reads a sequence of characters described by the diagram below. The signed real value of these characters is stored in the variable V. An error occurs if the sequence described below is not satisfied. If the value is outside the range of REAL or LONGREAL, variable V will contain the value infinity.



Type of V	**Conversion Rules**
STRING or PACKED ARRAY[0..n] OF CHAR	READ will fill V with characters from F while either EOLN(F) is FALSE or until V is full. It then sets the dynamic length of V (i.e. V[0]) to the number of characters read.

READDIR

Usage:

READDIR(F,I,V)
READDIR(F,I,V1, ... ,Vn)

Parameters:

F	must be a non-text file variable open in the read-write mode.
I	must be an expression compatible with SIGNED_16. It specifies the component number that will be read.
V	must be a variable. The component type of F must be compatible with the type of V. Any number of suitable variables may be specified.

Description:

The procedure READDIR(F,I,V) moves the current position of file F to component number I. It then assigns the value of that component to the variable V and advances the file position to the next component. An error occurs if the value of I is less than one or greater than MAXPOS(F).

```
READDIR(F,I,V)                is equivalent to  SEEK(F,I);
                                                GET(F);
                                                V := F^;
                                                GET(F);

READDIR(F,I,V1,V2, ... ,Vn) is equivalent to  READDIR(F,I,V1);
                                                READ(F,V2);
                                                ...
                                                READ(F,Vn);
```

Simulated I/O Note: The Simulated I/O Library does not fully implement read-write (i.e. random access) files. Refer to the section on Simulated I/O Library for important restrictions regarding positioning and writing on read-write files.

READDIR is an HP Standard Pascal extension of ANSI Pascal.

READLN

Usage:

READLN(F)
READLN(F,V)
READLN(F,V1, ... ,Vn)
READLN(V)
READLN(V1, ... ,Vn)
READLN

Parameters:

F	must be a textfile variable open in the read-only mode. If F is omitted or if the first parameter is not a file, the predefined variable INPUT is used.
V	must be a variable. The type of V must be compatible with CHAR, INTEGER, UNSIGNED__nn, REAL, LONGREAL, or STRING. V may be a component of a packed structure. Any number of suitable variables may be specified.

Description:

The procedure READLN(F) advances the current position of the textfile F to the beginning of the next line (i.e. to the first character following the next line marker). The procedure READLN(F,V) first reads characters from F and assigns their value to V before advancing to the next line. The rules assigning values to the variable V are the same as for READ(F,V).

```
READLN(F)                  is equivalent to  WHILE NOT EOLN(F) DO
                                             GET(F);
                                             GET(F);

READLN(F,V)                is equivalent to  READ(F,V);
                                             READLN(F);

READLN(F,V1,V2, ... ,Vn)   is equivalent to  READ(F,V1,V2, ... ,Vn);
                                             READLN(F);

READLN(V)                  is equivalent to  READ(INPUT,V);
                                             READLN(INPUT);

READLN(V1,V2, ... ,Vn)     is equivalent to  READ(INPUT,V1,V2, ... ,Vn);
                                             READLN(INPUT);

READLN                     is equivalent to  READLN(INPUT);
```

RESET

Usage:

```
RESET(F)
RESET(F,S)
RESET(F,S,T)
```

Parameters:

F	must be a file variable.
S	must be a string expression. The contents of S should be the name of a physical file that will be associated with F. The rules for physical file names and their meanings are not specified by Pascal. The interpretation of physical file names is done in the I/O environment library. Only the first 49 characters of S are significant.
T	must be a string expression. The contents of T specifies I/O implementation dependent information about the file F. The Pascal I/O library recognizes one value for T that specifies variable length record mode as opposed to fixed length records. 'VARIABLE' - specifies that physical records in the physical file will override the logical record description of the file variable. Other values for T will be ignored. The Pascal I/O library ignores leading and trailing blanks in T and considers upper and lower case letters equivalent.

Description:

The procedure RESET(F) opens the file F in the read-only state and places the current file position at the first component of the file. If the physical file associated with F does not exist, an error occurs. If the physical file is not empty, EOF(F) will return FALSE and a subsequent reference to F^ will actually load the buffer variable with the first component.

If the file F is already open when RESET is called, then F is first closed.

If the parameter S is specified and it is not the null string or all spaces, then the content of S is used as the physical file name to be associated with F. If S is not specified or if S is the null string or all spaces, then the name of the physical file previously associated with F, if F was open, is used. If F was not open and S is not specified, then the null string (i.e. '') is used as the physical file name.

Simulated I/O Note: The null string is not a legal file name for simulated I/O and causes an error.

REWRITE

Usage:

```
REWRITE(F)
REWRITE(F,S)
REWRITE(F,S,T)
```

Parameters:

F	must be a file variable.
S	must be a string expression. The contents of S should be the name of a physical file that will be associated with F. The rules for physical file names and their meanings are not specified by Pascal. The interpretation of physical file names is done in the I/O environment library. Only the first 49 characters of S are significant.
T	must be a string expression. The contents of T specifies I/O implementation dependent information about the file F. The Pascal I/O library recognizes one value for T that specifies variable length record mode as opposed to fixed length records. 'VARIABLE' - specifies that physical records in the physical file will override the logical record description of the file variable. Other values for T will be ignored. The Pascal I/O library ignores leading and trailing blanks in T and considers upper and lower case letters equivalent.

Description:

The procedure REWRITE(F) opens the file F in the write-only state and places the current file position at the first component of the file. If the physical file associated with F exists, it is deleted from the file system. Then the physical file is created. EOF(F) now returns TRUE and the contents of the buffer variable F^ is undefined.

If the file F is already open when REWRITE is called, then F is first closed.

If the parameter S is specified and it is not the null string or all spaces, then the content of S is used as the physical file name to be associated with F. If S is not specified or if S is the null string or all spaces, then the name of the physical file previously associated with F, if F was open, is used. If F was not open and S is not specified, then the null string (i.e. '') is used as the physical file name.

Simulated I/O Note: The null string is not a legal file name for simulated I/O and causes an error.

SEEK

Usage:

```
SEEK(F,I)
```

Parameters:

F	must be a non-text file variable open in the read-write state.
I	must be an expression compatible with SIGNED__16. It specifies the component number which will be the new file position.

Description:

The procedure SEEK(F,I) will place the current file position at component number I. If I is greater than the highest numbered component ever written to F, then EOF(F) will return TRUE. Otherwise EOF(F) will return FALSE. The buffer variable F^ is undefined. In order to load component I into the buffer variable F^, GET(F) must first be performed. An error will occur if I is less than one or greater than MAXPOS(F).

Simulated I/O Note: The Simulated I/O Library does not fully implement read-write (i.e. random access) files. Refer to the section on Simulated I/O Library for important restrictions regarding positioning and writing on read-write files.

SEEK is an HP Standard Pascal extension of ANSI Pascal.

STRREAD

Usage:

```
STRREAD(S,P,T,V)
STRREAD(S,P,T,V1, ... Vn)
```

Parameters:

S	must be a string expression.
P	must be an expression compatible with SIGNED__16.
T	must be a SIGNED__16 variable.
V	must be a variable compatible with CHAR, INTEGER, UNSIGNED__nn, REAL, LONGREAL, or STRING. Any number of suitable variables may be specified.

Description:

The procedure STRREAD(S,P,T,V) reads characters from string S, starting with the character S[P], and converts the characters into a value stored in V. After this operation, the variable T will contain the index into S, one greater than the last character that was read.

The rules for converting the characters in S to the value in V are the same as for READ(F,V) where F is a textfile. S is treated as a single line of a textfile. That is, an implicit line marker follows the last character in S.

An error occurs if an attempt is made to index beyond the current length of S (i.e. ORD(S[0])).

```
STRREAD(S,P,T,V1,V2, ... Vn) is equivalent to STRREAD(S,P,T,V1);
                                              STRREAD(S,T,T,V2);
                                              ...
                                              STRREAD(S,T,T,Vn);
```

STRREAD is an HP Standard Pascal extension of ANSI Pascal.

STRWRITE

Usage:

STRWRITE(S,P,T,E);
STRWRITE(S,P,T,E1, ... ,En)

Parameters:

S	must be a string variable.
P	must be an expression compatible with SIGNED__16.
T	must be a SIGNED__16 variable.
E	must be an expression. The type of E may be compatible with CHAR, INTEGER, UNSIGNED__nn, REAL, LONGREAL, or STRING. In addition, formatting information for the expression E may be specified using one of the three following forms:

E	{Specifies default formatting}
E:W	{W is SIGNED__16 compatible expression specifying field width}
E:W:N	{Only valid if E is type REAL or LONGREAL. W and N are} {SIGNED__16 compatible expressions. W specifies the field} {width and N specifies the number of digits written after} {the decimal point.}

Description:

The procedure STRWRITE(S,P,T,E) converts the value of expression E to a string of characters and writes those characters to string variable S starting with the character S[P]. After the operation, the variable T contains the index into S, one greater than the last character written. If, after the operation, the new length of S is greater than the previous length of S, then the length of S (i.e. ORD(S[0])) is updated to the new length.

The rules for converting the expression E are the same as for WRITE(F,E) where F is a textfile. An error occurs if an attempt is made to write beyond the maximum declared length of S. An error occurs if P is less than 1 or greater than ORD(S[0]) + 1.

```
STRWRITE(S,P,T,E1,E2, ... ,En) is equivalent to STRWRITE(S,P,T,E1);
                                                STRWRITE(S,T,T,E2);
                                                ...
                                                STRWRITE(S,T,T,En);
```

STRWRITE is an HP Standard Pascal extension of ANSI Pascal.

WRITE

Usage:

WRITE(F,E)
WRITE(F,E1, ... ,En)
WRITE(E)
WRITE(E1, ... ,En)

Parameters:

F	must be a file variable open in the write-only or read-write state. If the first parameter to WRITE is not a file, the predefined variable OUTPUT is used.
E	must be an expression. The type of E depends on whether F is a textfile.

If F is a non-text file, the type of E must be compatible with the component type of F.

If F is a textfile, the type of E may be compatible with CHAR, INTEGER, UNSIGNED__nn, REAL, LONGREAL, or STRING. In addition, formatting information for the expression E may be specified using one of the three following forms:

E	{Specifies default formatting}
E:W	{W is SIGNED__16 compatible expression specifying field width}

```
E:W:N  {Only valid if E is type REAL or LONGREAL.  W and N are}
       {SIGNED_16 compatible expressions.  W specifies the field}
       {width and N specifies the number of digits written after}
       {the decimal point.}
```

Description:

The procedure WRITE(F,E) writes the value of E to the current file component and advances the file position to the next component. The buffer variable F^ is undefined after WRITE.

```
WRITE(F,E1,E2, ... ,En) is equivalent to WRITE(F,E1);
                                         WRITE(F,E2);
                                         ...
                                         WRITE(F,En);

WRITE(E)                is equivalent to WRITE(OUTPUT,E);

WRITE(E1,E2, ... ,En)   is equivalent to WRITE(OUTPUT,E1);
                                         WRITE(OUTPUT,E2);
                                         ...
                                         WRITE(OUTPUT,En);
```

If F is a non-text file, the following equivalence describes the operation of WRITE.

```
WRITE(F,E)              is equivalent to F^ := E;
                                         PUT(F);
```

If F is a textfile, its components are of type CHAR. The expression E, however, may be of a type compatible with CHAR, INTEGER, UNSIGNED_nn, REAL, LONGREAL, or STRING. An implicit data conversion is performed from the internal represention of E to a character representation in the textfile F. The rules for the conversion depend on the type of E.

If the form of the expression is E:W or E:W:N, then the expression W specifies the total number of characters that will be written to textfile F in most cases. If W is not specified, then the width will default to the value given in the table below for the type of E. If W is greater than the number of characters needed to express E, then E will be preceded by spaces so that the total width is W. If W is less than the number needed to express E, then, for CHAR and STRING expressions, E will be truncated so that only the first W characters are written. For numeric expressions, W will be ignored so that the full significance of E will be written. An error will occur is the value of W or N is negative.

Before writing a field, WRITE checks if the field will fit on the present line (i.e. if W + LINEPOS(F) <= LINESIZE). If the field will not fit on the present line and the field will fit on a new line, then an automatic WRITELN is performed.

Type of E	Default Width	Conversion Rules
CHAR or subrange of CHAR	1	The value of CHAR expression E is written. No conversion is performed. If the value of W is zero, there is no output to the textfile.
INTEGER, BYTE, SIGNED__8, SIGNED__16, SIGNED__32, UNSIGNED__8 UNSIGNED__16, UNSIGNED__32, or subrange of above types	12	A string of decimal digits, possibly preceded by a minus sign, representing the value of E is written to the textfile. If the value of W is less than the number of digits required to represent E, then W is ignored so that no significance is lost.
REAL LONGREAL	12 20	For REAL or LONGREAL expressions, the programmer may specify N as well as W. If N is specified, WRITE writes E in a fixed-point format with N digits after the decimal point. If N is zero, the decimal point and subsequent digits are omitted. If N is not specified, WRITE writes E in a floating-point format consisting of a coefficient and a scale factor. In no case is it possible to write more significant digits than are contained in the internal representation of E. This implies that WRITE may change a fixed-point representation into a floating-point representation for some value of E.
STRING or PACKED ARRAY [0..N] OF CHAR	current length of string	If N (equal to ORD(E[0])) is the current length of string expression E, than N characters from E will be written to the textfile. If W is less than N, only the first W character will be written. If W is zero, there will be no output to the textfile.

WRITEDIR

Usage:

WRITEDIR(F,I,E)
WRITEDIR(F,I,E1, ... ,En)

Parameters:

F	must be a non-text file variable open in the read-write mode.
I	must be an expression compatible with SIGNED__16. It specifies the component number that will be written.
E	must be an expression. The component type of F must be compatible with the type of E. Any number of suitable expressions may be specified.

Description:

The procedure WRITEDIR(F,I,E) moves the current position of file F to component number I. It then assigns the value of expression E to that component and advances the file position to the next component. An error occurs if the value of I is less than one or greater than MAXPOS(F).

```
WRITEDIR(F,I,E)              is equivalent to  SEEK(F,I);
                                               F^ := E;
                                               PUT(F);

WRITEDIR(F,I,E1,E2, ... ,En) is equivalent to  WRITEDIR(F,I,E1);
                                               WRITE(F,E2);
                                               ...
                                               WRITE(F,En);
```

Simulated I/O Note: The Simulated I/O Library does not fully Implement read-write (i.e. random access) files. Refer to the section on Simulated I/O Library for important restrictions regarding positioning and writing on read-write files.

WRITEDIR is an HP Standard Pascal extension of ANSI Pascal.

WRITELN

Usage:

```
WRITELN(F)
WRITELN(F,E)
WRITELN(F,E1, ... ,En)
WRITELN(E)
WRITELN(E1, ... ,En)
WRITELN
```

Parameters:

F	must be a textfile variable open in the write-only mode. If F is omitted or if the first parameter is not a file, the predefined variable OUTPUT is used.
E	must be an expression. The type of E may be compatible with CHAR, INTEGER, UNSIGNED__nn, REAL, LONGREAL, or STRING. In addition, formatting information for the expression E may be specified using one of the three following forms:

```
E      {Specifies default formatting}

E:W    {W is SIGNED__16 compatible expression specifying field width}

E:W:N  {Only valid if E is type REAL or LONGREAL.  W and N are}
       {SIGNED__16 compatible expressions.  W specifies the field}
       {width and N specifies the number of digits written after}
       {the decimal point.}
```

Description:

The procedure WRITELN(F) writes a line marker to textfile F. The procedure WRITELN(F,E) first converts the expression E to textfile format and writes those characters to F before writing a line marker. The rules for the conversion of E are the same as for WRITE(F,E) for textfiles.

```
WRITELN(F,E)                 is equivalent to  WRITE(F,E);
                                               WRITELN(F);

WRITELN(F,E1,E2, ... ,En)    is equivalent to  WRITE(F,E1,E2, ... ,En);
                                               WRITELN(F);

WRITELN(E)                   is equivalent to  WRITE(OUTPUT,E);
                                               WRITELN(OUTPUT);

WRITELN(E1,E2, ... ,En)      is equivalent to  WRITE(OUTPUT,E1,E2, ... ,En);
                                               WRITELN(OUTPUT);

WRITELN                      is equivalent to  WRITELN(OUTPUT);
```

THE PASCAL I/O LIBRARY

The Pascal I/O Library is supplied in relocatable form with the Pascal/64000 compiler. Refer to your processor supplement manual for the specific file name of the library file or files.

Global Variable Definitions

The Pascal I/O Library defines the following global variables which may be useful to the programmer. In some cases they specify the behavior of features that are dependent on the I/O environment. Their default values are set by the procedure **Pinit__pascal__io** described below.

Global Variable Definition	Default Value	Explanation
INPUT:TEXT;	closed	The predefined file INPUT Is supplied by the Pascal I/O Library if it is used in the program.
OUTPUT:TEXT;	closed	The predefined file OUTPUT is supplied by the Pascal I/O Library if it Is used in the program.
Pbound: SIGNED__16;	32767	The maximum number of logical records that may be written in a non-text file. This Is the value returned by the function MAXPOS. An error occurs if an attempt is made to write or position to a record whose index is greater than Pbound.
LINESIZE: SIGNED__16;	256	The maximum number of characters that may be written to a line of a textfile. If the program attempts to write more than LINESIZE characters without calling for writing a line marker, an automatic WRITELN is performed.
LINE__MARKER__MODE :(REC,NL__CHAR);	REC	This variable describes the implementation of a line marker in a textfile. REC specifies that a physical record boundary constitutes a line marker. NL__CHAR specifies that a particular character, contained In the variable LINE__MARKER__CHAR, constitutes a line marker.

LINE__MARKER__CHAR :CHAR;	CHR(10) ASCII linefeed	If LINE__MARKER__MODE equals NL__CHAR, then the value of LINE__MARKER__CHAR is the character that constitutes a line marker.
PAD__ODD__RECORDS: BOOLEAN;	TRUE	For non-text file variables, TRUE specifies that, for files whose components are an odd number of bytes, the component length will be adjusted upward to an even number of bytes. When writing, an extra space character will be added to the logical record. This feature allows consistant operation in the Simulated I/O environment where physical disc records must contain an even number of bytes.
READ__REC__LEN: SIGNED__16;	none	Whenever an input operation is performed, READ__REC__LEN is set to contain the number of data bytes transferred from the physical file to the buffer variable. In variable length record mode, this is either the length of the physical record or the length of the file component.
WRITE__REC__LEN: SIGNED__16;	0	For files in variable length record mode, WRITE__REC__LEN, if non-zero, specifies the number of data bytes to transfer from the buffer variable to the file on the next output operation. If WRITE__REC__LEN equals zero or if the file is not in 'VARIABLE' mode then the number of bytes transfer red is equal to the length of the file component type. After all output operations, WRITE__REC__LEN is set to zero.
Piocheck:BOOLEAN;	none	Every call to a routine in the Pascal I/O Library generated by the Pascal/64000 compiler contains a BOOLEAN parameter which specifies the value of $IOCHECK$ compiler option. This parameter is transferred to Piocheck for every

		I/O operation for use by the error handling routine, Perror.
Preturn__addr: ^BYTE;	none	Outer level routines in the Pascal I/O Library save the return address of their caller in Preturn__addr. It may be used by the error handling routine, Perror.
IOR:SIGNED__8;	none	IOR contains the result code of the most recent I/O operation. It is referenced by the function IORESULT.

Description of the Pascal I/O Library Routines

The programmer may call the following routines explicitly from the Pascal program. They are not predefined. They must be declared EXTERNAL in the program.

PROCEDURE Pinit__pascal__io;

This routine must be called by the programmer before any I/O operations are performed. It initializes the data used by the other Pascal I/O Library routines to a known state.

PROCEDURE Pshutdown__pascal__io;

This routine may be called by the programmer at the end of a Pascal program to close any files that remain open.

FUNCTION IORESULT: SIGNED__8;

This function may be called by the programmer to obtain a result code for the most recent I/O operation. The result code is interpreted as follows:

0 - No error, previous operation was successful.

1 - The I/O environment library detected an error. See the I/O environment documentation for further information.

2 - File is closed.

3 - File is not open in proper mode. That is, input operation on write-only file, output operation on read-only file, or direct access operation on read-only or write-only file.

4	-	Input operation done when end-of-file is TRUE
5	-	When reading a number from a textfile, the characters did not form a proper number of the desired type.
6	-	When reading a number from a textfile, a number was found but its value was outside the range of internal representation.
7	-	A textfile width parameter (e.g. E:WIDTH) had a value less than zero.
8	-	Too many records in the file. The maximum number of records is in the global variable Pbound.
9	-	During a STRWRITE or STRREAD, the starting string index was less than 1, greater than the present length of the string + 1, or greater than the maximum length of the string.
10	-	During a SEEK, READDIR, or WRITEDIR operation the desired record could not be found. Either the record number was less than 1 or the record did not exist.

PROCEDURE Perror(ERRCODE:SIGNED__8);

Perror is not normally called by the programmer. It is the error handling routine called by the Pascal I/O Library. ERRCODE is a value interpreted as shown for IORESULT above. If the global variable **Piocheck** is TRUE, then **Perror** produces a run-time error in a processor dependent manner. When moving out of the emulation environment, the programmer may want to provide his own version of **Perror**.

Library Routines Called by the Pascal/64000 Compiler

The following routines are called by the Pascal/64000 compiler to implement the various Pascal I/O features.

Most of these routines make reference to a file variable. The descriptions below use the record type FIB (for File Information Block) to describe a file variable. Every file variable contains a buffer variable. For a variable F: FILE OF T, the buffer variable is of type T and can be of any length. To implement this, the last field of the FIB record, BUFVAR, is the buffer variable. Although BUFVAR is declared as a long array, this is merely a definition of convenience. The Pascal/64000 compiler only allocates enough memory to accomodate a buffer variable of type T, whatever size that may be.

```
FIB = RECORD
        ...
        {VARIOUS FIELDS USED BY PASCAL I/O LIBRARY}
        ...
        {The last field implements the buffer variable.  In practice,}
        {the compiler does not allocate 32k bytes for this field but }
```

```
          {only enough to hold the files component type. }
          BUFVAR:ARRAY[0..32767] OF BYTE;
          END;
```

```
PROCEDURE Pappend(VAR F:FIB; VAR NAME,INFO:STRING; SIZE:SIGNED_16;
                  ISTEXT,IOCHECK:BOOLEAN);
```

Pappend implements APPEND. Parameters:

```
     F         - File variable.
     NAME      - Physical file name.  If parameter omitted in source,
                 compiler generates null string (i.e. '').
     INFO      - I/O environment information.  If parameter omitted in
                 source, compiler generates null string (i.e. '');
     SIZE      - Size, in bytes, of file component.
     ISTEXT    - TRUE if textfile.
     IOCHECK   - Value of $IOCHECK$ compiler option.
```

```
FUNCTION  Pbufvar(VAR F:FIB; CONTINUED,IOCHECK:BOOLEAN): ^BYTE;
```

Pbufvar implements references to the buffer variable F^. It returns the address of the file's buffer variable. Under certain circumstances, Pbufvar performs an input operation. Parameters:

```
     F         - File variable.
     CONTINUED - TRUE indicates that this call is the continuation of
                 a sequence of calls that comprise a single operation.
                 If TRUE and a previous call in the sequence detected
                 an error, then no operation is performed.
     IOCHECK   - Value of the $IOCHECK$ compiler option.
```

```
PROCEDURE Pclose(VAR F:FIB; VAR INFO:STRING; IOCHECK:BOOLEAN);
```

Pclose implements CLOSE. Parameters:

```
     F         - File variable.
     INFO      - Information regarding file disposition.  If parameter
                 omitted in source, compiler generates null string
                 (i.e. ('')).
     IOCHECK   - Value of $IOCHECK$ compiler option.
```

```
FUNCTION  Peof(VAR F:FIB; IOCHECK:BOOLEAN): BOOLEAN;
```

Peof implements EOF. Under certain circumstances Peof performs an input operation. Parameters:

```
     F         - File variable.
     IOCHECK   - Value of $IOCHECK$ compiler option.
```

```
FUNCTION  Peoln(VAR F:FIB; IOCHECK:BOOLEAN): BOOLEAN;
```

Peoln implements EOLN. Under certain circumstances Peoln performs an input operation. Parameters:

```
      F         - File variable.
      IOCHECK   - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Pget(VAR F:FIB; CONTINUED,IOCHECK:BOOLEAN);
```

Pget implements GET. Under certain circumstances Pget performs an input operation. Parameters:

```
      F         - File variable.
      CONTINUED - TRUE indicates that this call is the continuation of
                  a sequence of calls that comprise a single operation.
                  If TRUE and a previous call in the sequence detected
                  an error, then no operation is performed.
      IOCHECK   - Value of the $IOCHECK$ compiler option.
```

```
FUNCTION  Plinepos(VAR F:FIB; IOCHECK:BOOLEAN): SIGNED_16;
```

Plinepos implements LINEPOS. Parameters:

```
      F         - File variable.
      IOCHECK   - Value of $IOCHECK$ compiler option.
```

```
FUNCTION  Pmaxpos(VAR F:FIB; IOCHECK:BOOLEAN): SIGNED_16;
```

Pmaxpos implements MAXPOS. Parameters:

```
      F         - File variable.
      IOCHECK   - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Popen(VAR F:FIB; VAR NAME,INFO:STRING; SIZE:SIGNED_16;
              ISTEXT,IOCHECK:BOOLEAN);
```

Popen implements OPEN. Parameters:

```
      F         - File variable.
      NAME      - Physical file name. If parameter omitted in source,
                  compiler generates null string (i.e. '').
      INFO      - I/O environment information. If parameter omitted in
                  source, compiler generates null string (i.e. '');
      SIZE      - Size, in bytes, of file component.
      ISTEXT    - TRUE if textfile.
      IOCHECK   - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Poverprint(VAR F:FIB; IOCHECK:BOOLEAN);

   Poverprint implements OVERPRINT.  Parameters:

      F          - File variable.
      IOCHECK    - Value of $IOCHECK$ compiler option.


PROCEDURE Ppage(VAR F:FIB; IOCHECK:BOOLEAN);

   Ppage implements PAGE.  Parameters:

      F          - File variable.
      IOCHECK    - Value of $IOCHECK$ compiler option.


FUNCTION  Pposition(VAR F:FIB; IOCHECK:BOOLEAN): SIGNED_16;

   Pposition implements POSITION. Parameters:

      F          - File variable.
      IOCHECK    - Value of $IOCHECK$ compiler option.


PROCEDURE Pprompt(VAR F:FIB; IOCHECK:BOOLEAN);

   Pprompt implements PROMPT.  Parameters:

      F          - File variable.
      IOCHECK    - Value of $IOCHECK$ compiler option.


PROCEDURE Pput(VAR F:FIB; CONTINUED,IOCHECK:BOOLEAN);

   Pput implement PUT.  Parameters:

      F          - File variable.
      CONTINUED  - TRUE indicates that this call is the continuation of
                   a sequence of calls that comprise a single operation.
                   If TRUE and a previous call in the sequence detected
                   an error, then no operation is performed.
      IOCHECK    - Value of the $IOCHECK$ compiler option.


FUNCTION  Pread_char(VAR F:FIB; IOCHECK:BOOLEAN): CHAR;

   Pread_char implements READ(F,V) where F is a textfile and V is
compatible with CHAR.  Parameters:

      F          - File variable.
      IOCHECK    - Value of $IOCHECK$ compiler option.
```

```
FUNCTION  Pread_integer(VAR F:FIB; IOCHECK:BOOLEAN): INTEGER;
```

Pread_integer implements READ(F,V) where F is a textfile and V is compatible with INTEGER. Parameters:

F	- File variable.
IOCHECK	- Value of $IOCHECK$ compiler option.

```
PROCEDURE Pread_longreal(VAR F:FIB; VAR L:LONGREAL; IOCHECK:BOOLEAN);
```

Pread_longreal implements READ(F,V) where F is a textfile and V is type LONGREAL. Parameters:

F	- File variable.
L	- LONGREAL variable.
IOCHECK	- Value of $IOCHECK$ compiler option.

```
PROCEDURE Pread_real(VAR F:FIB; VAR R:REAL; IOCHECK:BOOLEAN);
```

Pread_real implements READ(F,V) where F is a textfile and V is type REAL. Parameters:

F	- File variable.
R	- REAL variable.
IOCHECK	- Value of $IOCHECK$ compiler option.

```
PROCEDURE Pread_string(VAR F:FIB; VAR S:STRING; MAXLEN:SIGNED_16;
            IOCHECK:BOOLEAN);
```

Pread_string implements READ(F,V) where F is a textfile and V is compatible with STRING. Parameters:

F	- File variable.
S	- STRING variable.
MAXLEN	- Maximum declared length of string variable.
IOCHECK	- Value of $IOCHECK$ compiler option.

```
FUNCTION  Pread_unsigned(VAR F:FIB; IOCHECK:BOOLEAN): UNSIGNED_nn;
```

Pread_unsigned implements READ(F,V) where F is a textfile and V is compatible with UNSIGNED_nn. UNSIGNED_nn is defined to be UNSIGNED_16 for the 6800 and 6809 processors and UNSIGNED_32 for other processors.

F	- File variable.
IOCHECK	- Value of $IOCHECK$ compiler option.

```
PROCEDURE Preadln(VAR F:FIB; IOCHECK:BOOLEAN);

   Preadln implements READLN.  Parameters:

      F          - File variable.
      IOCHECK    - Value of $IOCHECK$ compiler option.


PROCEDURE Preset(VAR F:FIB; VAR NAME,INFO:STRING; SIZE:SIGNED_16;
             ISTEXT,IOCHECK:BOOLEAN);

   Preset implements RESET.  Parameters:
      F          - File variable.
      NAME       - Physical file name.  If parameter omitted in source,
                   compiler generates null string (i.e. '').
      INFO       - I/O environment information.  If parameter omitted in
                   source, compiler generates null string (i.e. '');
      SIZE       - Size, in bytes, of file component.
      ISTEXT     - TRUE if textfile.
      IOCHECK    - Value of $IOCHECK$ compiler option.


PROCEDURE Prewrite(VAR F:FIB; VAR NAME,INFO:STRING; SIZE:SIGNED_16;
                   ISTEXT,IOCHECK:BOOLEAN);

   Prewrite implements REWRITE.  Parameters:
      F          - File variable.
      NAME       - Physical file name.  If parameter omitted in source,
                   compiler generates null string (i.e. '').
      INFO       - I/O environment information.  If parameter omitted in
                   source, compiler generates null string (i.e. '');
      SIZE       - Size, in bytes, of file component.
      ISTEXT     - TRUE if textfile.
      IOCHECK    - Value of $IOCHECK$ compiler option.


PROCEDURE Pseek(VAR F:FIB; N:SIGNED_16; IOCHECK:BOOLEAN);

   Pseek implements SEEK.  Parameters:

      F          - File variable.
      N          - Desired component number.
      IOCHECK    - Value of $IOCHECK$ compiler option.


PROCEDURE Pstringopen(VAR S:STRING; MAXLEN,P: SIGNED_16;
             VAR T:SIGNED_16; IOCHECK:BOOLEAN);
```

Pstringopen implements STRREAD(S,P,T, ...) and STRWRITE(S,P,T, ...). It initializes an internal textfile variable, STRfile, with information that directs input and output to that file variable to a string instead of a physical file. Subsequent calls to I/O routines refer to STRfile. Parameters:

```
    S          - STRING expression.
    MAXLEN     - Maximum declared length of S.
    P          - Starting index value.
    T          - Variable holding ending index value.
    IOCHECK    - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Pwriteln(VAR F:FIB; IOCHECK:BOOLEAN);
```

Pwriteln implements WRITELN. Parameters:

```
    F          - File variable.
    IOCHECK    - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Pwrite_char(VAR F:FIB; C:CHAR; WIDTH:SIGNED_16; IOCHECK:BOOLEAN);
```

Pwrite_char implements WRITE(F,E) where F is a textfile and E is compatible with CHAR. Parameters:

```
    F          - File variable.
    C          - CHAR expression.
    WIDTH      - Total width of field.  If omitted in source, the
                 compiler generates the value 1.
    IOCHECK    - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Pwrite_integer(VAR F:FIB; I:INTEGER; WIDTH:SIGNED_16;
              IOCHECK:BOOLEAN);
```

Pwrite_integer implements WRITE(F,E) where F is a textfile and E is compatible with INTEGER. Parameters:

```
    F          - File variable.
    I          - INTEGER expression.
    WIDTH      - Total width of field.  If omitted in source, the
                 compiler generates the value 12.
    IOCHECK    - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Pwrite_longreal(VAR F:FIB; L:LONGREAL; WIDTH,N:SIGNED_16;
              IOCHECK:BOOLEAN);
```

Pwrite_longreal implements WRITE(F,E) where F is a textfile and E is type LONGREAL. Parameters:

```
    F          - File variable.
    L          - LONGREAL expression.
    WIDTH      - Total width of field.  If omitted in source, the
                 compiler generates the value 20.
    N          - Number digits after decimal point.  If omitted in
                 source, the compiler generates the value 9999.
    IOCHECK    - Value of $IOCHECK$ compiler option.
```

```
PROCEDURE Pwrite_real(VAR F:FIB; R:REAL; WIDTH,N:SIGNED_16; IOCHECK:BOOLEAN);
```

Pwrite_real implements WRITE(F,E) where F is a textfile and E is type REAL. Parameters:

F	- File variable.
R	- REAL expression.
WIDTH	- Total width of field. If omitted in source, the compiler generates the value 12.
N	- Number digits after decimal point. If omitted in source, the compiler generates the value 9999.
IOCHECK	- Value of $IOCHECK$ compiler option.

```
PROCEDURE Pwrite_string(VAR F:FIB; VAR S:STRING; WIDTH:SIGNED_16;
            IOCHECK:BOOLEAN);
```

Pwrite_string implements WRITE(F,E) where F is a textfile and E is compatible with STRING. Parameters:

F	- File variable.
I	- STRING expression.
WIDTH	- Total width of field. If omitted in source, the compiler generates the value -9999.
IOCHECK	- Value of $IOCHECK$ compiler option.

```
PROCEDURE Pwrite_unsigned(VAR F:FIB; U:UNSIGNED_nn; WIDTH:SIGNED_16;
            IOCHECK:BOOLEAN);
```

Pwrite_unsigned implements WRITE(F,E) where F is a textfile and E is compatible with UNSIGNED_nn. UNSIGNED_nn is defined to be UNSIGNED_16 for the 6800 and 6809 processor and UNSIGNED_32 for other processors.

F	- File variable.
I	- INTEGER expression.
WIDTH	- Total width of field. If omitted in source, the compiler generates the value 12.
IOCHECK	- Value of $IOCHECK$ compiler option.

SIMULATED I/O LIBRARY

The Simulated I/O Library is supplied in relocatable form with the Pascal/64000. Refer to the processor supplement manual for the names of the library file or files.

Description of Simulated I/O Files and Devices

The Simulated I/O Library allows a program running in the emulation environment to access a number of physical files simultaneously. The maximum number of physical files that may be open at one time is limited by the amount of buffer memory available in the HP 64000 host processor. The amount of buffer memory available is a function of the number of measurement system options that are installed in the HP 64000 card cage. The amount of buffer memory that is available is as follows:

One meas__sys module	768 words
Two meas__sys modules	512 words
Three meas__sys modules	256 words
Four meas__sys modules	0 words

The amount of buffer space required for each physical file depends on the device type and is as follows:

Printer	145 words
display	145 words
rs232	128 words plus an additional 128 words if the rs232 is open for reading.
keyboard	0 words
disc file	145 words

If, when opening a file, the host processor cannot allocate a buffer of the required size, an error message will occur and the global variable **errno** will be set to 9.

A description of the physical files is given in the following paragraphs.

PRINTER. The 64000 printer is identified with the physical file name 'printer'. It may only be opened in write-only mode. The printer has a maximum line width of 132 characters. If longer lines are written, the data beyond 132 characters will be lost.

It is possible for several logical files to open the printer simultaneously. In this case, the data written from the several files will be mixed on the page.

DISPLAY. The 64000 display is identified with the physical file name 'display'. It may only be opened in the write-only mode. The display consists of 18 lines of 80 characters.

There are two modes of writing to the display. In the default mode, the display is rolled up and new data is written to the 18th line. If there are more than 80 characters in the line, data beyond the 80th character is lost. Alternatively, the programmer may select a starting line and column and write up to 255 characters beginning at that line and column. In this mode, data beyond the 80th column is written on the next line.

Two global variables, defined in the Simulated I/O Library, control the display mode.

```
    DSP_LINE:   1..18; {Beginning line number for next write}
    DSP_COLUMN: 1..80; {Beginning column number for next write}
```

The routine INIT__SIMIO__LIB performs initialization as follows:

```
   DSP_LINE := 18;
   DSP_COLUMN := 1;
```

If DSP__LINE equals 18 and DSP__COLUMN equals 1, then the roll up mode is selected. If these variables have any other value, then the random access mode is selected.

NOTE : Setting the display line and column is not the equivalent of performing a WRITELN. The Pascal I/O Library performs a WRITELN every 256 characters if one was not executed by the Pascal program. Programmers using random access display modes and no WRITELNs obtain surprising results.

It is possible for several logical files to open the display simultaneously. In this case, the data written from the several files will be mixed on the screen.

KEYBOARD. The 64000 keyboard is identified with the physical file name 'keyboard'. It may only be opened in the read-only mode. A maximum of 240 characters may be entered into the command line area at the bottom of the 64000 display. An end-of-file condition is produced by entering a zero length line.

The keyboard operates in two modes. In the default mode, only the (RETURN) key terminates keyboard input and causes data to be returned to the caller. Alternatively, the programmer may select a mode where there are a number of terminating conditions. For instance, the (TAB) key, (ROLL UP) key, a full line, or others described below cause data to be returned to the caller.

The following global variables, defined in the Simulated I/O Library, control the operation of the keyboard.

```
KBD_ALL_TERMS: BOOLEAN; {FALSE==>only RETURN key works.}
                        {TRUE ==> all terminators work}
KBD_CMND_CODE: -2..-1;  {-2==>clear line on 1st key, -1==>retain former line}
KBD_LINE_LEN:  1..240;  {Specifies maximum keyboard read length}
KBD_TERM_CODE: 8..24;   {Returns terminator code from last keyboard read}
KBD_LOST_CHAR: CHAR;    {Returns "lost" char for certain}
                        {terminating conditions}
```

The routine INIT__SIMIO__LIB performs initialization as follows:

```
KBD_ALL_TERMS := FALSE;
KBD_CMND_CODE := -2;
KBD_LINE_LEN  := 240;
```

KBD__TERM__CODE is always set after a keyboard read to indicate the condition which terminated the read. KBD__LOST__CHAR may be set after certain terminating conditions. The values of KBD__TERM__CODE have the following meanings. Note that if KBD__ALL__TERMS is FALSE, KBD__TERM__CODE will only contain the value 13.

```
8  - While in INSERT CHAR mode, a character was typed into a full
     line. KBD_LOST_CHAR contains the character that was "lost"
     from the right of the line.
9  - The TAB key was pressed.
10 - The DOWN ARROW key was pressed.
11 - The UP ARROW key was pressed.
12 - The NEXT PAGE key was pressed.
13 - The RETURN key was pressed.
14 - An attempt was made to move the cursor right, past the end of
     the line.
15 - An attempt was made to move the cursor left, past the beginning
     of the line.
16 - The DELETE CHAR key was pressed deleting a character from a
     previously full line.
17 - The shifted TAB key was pressed.
18 - The PREV PAGE key was pressed.
19 - The ROLL DOWN key was pressed.
20 - The ROLL UP key was pressed.
21 - The shifted RIGHT ARROW key was pressed.
22 - The shifted LEFT ARROW key was pressed.
23 - The CLR LINE key was pressed.
24 - While in INSERT CHAR mode, a character was typed when the
     cursor was positioned at the end of the line. KBD_LOST_CHAR
     contains the character that was typed.
```

It is possible for several logical files to open the keyboard simultaneously. In this case, the several files will obtain data from the keyboard sequentially in the order of their respective reads.

RS232. The HP 64000 RS232 port is identified with the physical file name 'rs232'. It may be opened in the read-only, write-only, or read-write state. It can send and receive data in the asynchronous mode only.

The following characteristics of the rs232 port are selected by switches on the HP 64000 I/O board. Refer to the HP 64000 System Software Reference Manual for a more detailed description on the following:

RS232 or current loop interface
20 mA or 60 mA, current loop
Asynchronous baud rate
Internal or external clock input

The following characteristics of the rs232 may be specified by the user's program at run-time:

Number of data bits per character (5, 6, 7, or 8)
Character parity (none, odd, or even)
Number of stop bits (1, 1.5, or 2)

The following global variable, defined by the simulated I/O Library, is used to specify characteristics listed above:

```
RS232__MODE__BYTE: BYTE;
```

The routine INIT__SIMIO__LIB performs initialization as follows:

```
RS232_MODE_BYTE := 01111010B; {Select 1 stop bit, even parity,  }
                              {7 data bits, 1/16 times Tx clock }
```

After calling INIT__SIMIO__LIB and before opening the rs232 device, the user may change the value of RS232__MODE__BYTE. The bits of RS232__MODE__BYTE have the following meaning:

bit	7	6	5	4	3	2	1	0
identifier	S2	S1	EP	PE	L2	L1	B2	B1

where:

S2	S1	transmitted stop bits
0	0	invalid
0	1	1
1	0	1.5
1	1	2

EP(even parity), 1 ==> even parity, 0 ==> odd parity

PE(parity enable), 1 ==> parity bit enable, 0 ==> no parity bit

L2	L1	data bits (excluding parity)
0	0	5
0	1	6
1	0	7
1	1	8

B2	B1	clock mode
0	0	synchronous mode (not supported)
0	1	baud rate = 1 times clock rate
1	0	baud rate = 1/16 times clock rate (normal setting)
1	1	baud rate = 1/64 times clock rate (not supported)

It is useful to think of the rs232 port as two devices, a transmitter and a receiver, that share the same hardware resource but operate independently. Both the transmitter and receiver use buffers that are a part of the 240-byte rs232 CA buffer. There is a 100-byte buffer space used by the transmitter and a 100-byte buffer space used by the receiver.

The rs232 device is transparent to all character sets and protocols. The Simulated I/O Library does not detect or automatically generate any control characters. Detection and generation of control characters must be accomplished by the user's program.

The end-of-file (eof) condition is undefined for the rs232 device. The function **read**, when applied to the rs232 device, will never return an end-of-file indication.

The Simulated I/O Library does not detect character parity errors, character overrun errors, character framing errors, or buffer overflow errors. In order to ensure correct reception of data, the user must employ some method of error detection (e.g. check sums).

When applied to the rs232 device, the Simulated I/O Library function **write** operates as follows:

(Assume execution of the following statement)

```
I := write(FD,BUFFER,N,REC__BOUND);
```

then:

a. **write** begins to transfer N data bytes from the location specified by BUFFER to the 100-byte rs232 transmitter buffer.

b. If the 100-byte transmitter buffer becomes full, a command is issued to initiate the transmission of the data in the 100-byte buffer. When this is complete, the buffer is considered to be empty. The transfer of data from the user's buffer to the transmitter buffer resumes.

c. After N bytes of data have been transferred to the transmitter buffer, **write** checks the value of REC__BOUND. If REC__BOUND is zero, **write** simply returns. Any data in the transmitter buffer is stored until a subsequent call to **write**. If REC__BOUND is non-zero, a command is issued to initiate the transmission of any data in the transmitter buffer. When this is done, the buffer is considered empty.

When applied to the rs232 device, the function **read** operates as follows (assume execution of the following statement):

I := read(FD,BUFFER,N,REC__BOUND)

then:

a. **read** checks the number of data characters in the 100-byte receiver buffer. If there are any data characters, skip to subparagraph c below. If there is no data available, proceed to subparagraph b.

b. **read** issues a command to transfer data from the HP 64000 read buffer to the 100-byte receiver buffer. This will include all data received by the rs232 device since the last **read** command. It is possible that no data will be transferred if none was received during this interval.

   It is also possible that more than 100 bytes of data were received since the last update command was issued. In this case, some of the received data will be lost and no indication of the loss will be given. The user must ensure that input operations are done frequently enough so that all data is received.

c. **read** decides how many bytes to transfer to BUFFER. If N is less than the number of bytes in the 100-byte receiver buffer, then N characters will be transferred. If N is greater than or equal to the available data, then all the available data will be transferred and REC__BOUND will be set to TRUE. The proper number of bytes, which may be zero, are transferred to BUFFER and the count of data bytes available in the receiver buffer is reduced.

When programming in Pascal/64000, the user does not call the Simulated I/O Library routines **read** and **write** directly. These routines are called by the Pascal I/O Library routines. The behavior of Pascal programs using rs232 is described in the following paragraphs.

In Pascal/64000, the user may open a textfile to the rs232 transmitter for writing and open another textfile to the rs232 receiver for reading. In this instance, the Pascal I/O Library global, LINE__MARKER__MODE, should be REC, the default case, specifing that physical record boundaries constitute line markers. When this is done, line markers and physical records are the same and the following programming methods may be used.

When transmitting, the procedure WRITELN specifies that a physical record boundary (i.e. line marker) be written and this initiates transmission of any data stored in the 100-byte transmitter buffer. For example:

```
VAR TRANSMITTER: TEXT;
BEGIN
{Initialization of libraries not shown}
REWRITE(TRANSMITTER,'rs232');
WRITE(TRANSMITTER,'Some data'); {'Some data' is stored in transmitter buffer}
WRITELN(TRANSMITTER);           {This statement initiates transmission of   }
                                {data in the transmitter buffer.            }
```

When receiving, the function EOLN is used to determine if any received data is available. When no data is available, EOLN returns TRUE. This is because the Simulated I/O function

read detects a physical record boundary when data is unavailable and physical record boundaries are the same as line markers. For example:

```
VAR RECEIVER: TEXT;  C:CHAR;
BEGIN
{Initialization of libraries not shown}
RESET(RECEIVER,'rs232');
WHILE TRUE DO
  IF EOLN(RECEIVER) THEN
    READLN(RECEIVER)   {Data is temporarily unavailable. Skip to line marker}
  ELSE
    BEGIN
    READ(RECEIVER,C);  {Get a character from the rs232 device }
    {Process the character}
    END;
```

Another method of using the rs232 device employs non-text files in variable length record mode. The Pascal I/O Library global variables WRITE__REC__LEN and READ__REC__LEN are used to specify and determine the length of physical records which are the same as actual transmissions and receptions. For example:

```
TYPE
   RS232_REC = ARRAY[1..100] OF CHAR;
VAR
   RECEIVER    :FILE OF RS232_REC;
   TRANSMITTER :FILE OF RS232_REC;
   RECV DATA   :RS232 REC;
   SEND_DATA   :PACKED ARRAY[0..100] OF CHAR; {Data to transmit}
   I           :SIGNED_16
   ETX         :BOOLEAN;       {TRUE indicates end of received data}
$EXTVAR ON$
   READ_REC_LEN   :SIGNED_16;  {No. of bytes in last reception}
   WRITE_REC_LEN  :SIGNED_16;  {No. of bytes in next transmission}
$EXTVAR OFF$
   BEGIN
   {Initialization of libraries not shown}
   RESET(RECEIVER,'rs232','VARIABLE');
   REWRITE(TRANSMITTER,'rs232','VARIABLE');
   SEND_DATA := 'Here is some data'; {Get some data to send}
   WRITE_REC_LEN := ORD(SEND_DATA[0]); {Set length of transmission}
   FOR I := 1 TO WRITE_REC_LEN DO  {Transfer data to buffer variable}
     TRANSMITTER^[I] := SEND_DATA[I];
   PUT(TRANSMITTER);             {Initiate transmission}
   ETX := FALSE;
   WHILE NOT ETX DO
     READ(RECEIVER,RECV_DATA);   {Read rs232 data}
     FOR I := 1 TO READ_REC_LEN DO  {If READ_REC_LEN = 0, do nothing}
       IF RECV_DATA[I] = CHR(3) THEN
         ETX := TRUE                {Terminator found}
       ELSE
         {Process received data};
     END;  {Receive loop}
```

DISC FILES. A maximum of six 64000 disc files may be open simultaneously. These files may be opened in read-only, write-only, or read-write mode. The physical file name of a disc file has the following syntax.

```
<NAME> [:<USERID>] [:<DISC>] [:<TYPE>]
```

where:

<NAME>	is an identifier beginning with an upper case letter. The subsequent characters may be upper or lower case letters, digits, or the underscore character. Only the first nine characters are significant.
<USERID>	is an identifier beginning with an upper case letter. The subsequent characters may be upper or lower case letters, digits, or the underscore character. Only the first six characters are significant. Note, simulated I/O does not accept the blank userid.
<DISC>	is a number in the range 0 through 7.
<TYPE>	is one of the following: source reloc absolute listing emul_com link_com trace prom data asmb_db asmb_sym link_sym comp_sym comp_db

Disc files are composed of variable length physical records. A file may contain from 0 to 32767 physical records. Each physical record may contain 2 to 256 bytes of data and always consists of an even number of bytes.

NOTE: If several logical files open the same disc file simultaneously, the Simulated I/O Library may produce unexpected results. Generally, it is alright if several files access the same disc file for input only. However, the user is warned that file data will be corrupted if several files open the same disc file for output.

Random Access Restrictions

The 64000 file system is inherently sequential. It does not allow truly random access to records. More important, it does not allow a record in the middle of a file to be rewritten. The programmer using read-write files should be aware of the following points:

a. All record accesses are sequential. In order to access a particular record, simulated I/O positions to the beginning of the file and starts reading until the desired record is found. This implies very slow performance for seek operations.

b. After every write operation to a disc file, the end-of-file mark is always written. If one rewrites a record in the middle of a disc file, all the records which follow will be irretrievably lost.

   In the default mode, the Simulated I/O Library disallows write operations to disc files open in the read-write mode. This is so that a programmer will not inadvertantly lose file data. Writes to read-write files may be enabled using the following global variable defined in the Simulated I/O Library.

```
RW__WRITE__ENABLE: BOOLEAN; {FALSE causes writes to read-write files
                            to produce an error}
```

   The routine INIT__SIMIO__LIB performs initialization as follows:

```
RW__WRITE__ENABLE := FALSE;
```

   The programmer may enable writes in read-write mode using the variable above. In practice, this only makes sense if he or she intends to add records to the end of a file.

c. The Simulated I/O Library will not place the file position beyond the end-of-file mark. Programmers using random access are accustomed to writing records in a random order. This is impossible with simulated I/O. An error will occur if an attempt is to position to a record beyond the end-of-file mark which exists after the last existing record in the file.

Defining CA Buffers

The 64000 Simulated I/O facility uses buffers, called CA buffers, to communicate between the emulation and host environments. The programmer must reserve memory for whatever CA buffers his program will use. In addition, the addresses of the CA buffers must be given to the Simulated I/O Library routine, INIT__SIMIO__LIB. The same addresses must be given during emulation configuration to the Simulated I/O Configuration.

The length of the CA buffers for various devices is given below. If the program will not use a particular device, then no CA buffer need be defined for that device.

```
display   - 257 bytes
printer   - 242 bytes
rs232     - 240 bytes
keyboard  - 243 bytes
disc file - 258 bytes.  Up to six disc file CA buffers may be
            defined.  The number of disc file CA buffers that need
            to be defined equals the maximum number of disc
            files that will be open at one time.
```

The programmer will find it convenient to "ORG" the CA buffers at an absolute memory address. This is because the absolute address of the buffers must be specified during emulation configuration. Using the ORG option prevents the CA buffers from moving in memory when the program is changed and eliminates the need to change the emulation configuration.

Error Reporting

All the routines in the Simulated I/O Library are functions returning SIGNED_16 values. The value -1 (and other values for some functions) indicates that an error has occured. When an error occurs, the global variable **errno** is set to indicate the nature of the error and the function returns a value indicating an error.

The Simulated I/O Library defines **errno** and its values as follows: In the list below, the comment "Simio" indicates an error code returned from an actual simulated I/O command. The comment "Soft" indicates an error detected by the Simulated I/O Library software.

```
   errno: SIGNED_16;

 0 - Simio - No error. Operation was successful.
 1 - Simio - End of file.  This is not an error condition.
 2 - Simio - Invalid disc number.
 3 - Simio - File not found.
 4 - Simio - File already exists.
 5 - Simio - Insufficient disc space.
 6 - Simio - Directory full.
 7 - Simio - Corrupt file linkage.
 8 - Simio - Error reading or writing CA buffer.
 9 - Simio - Simio request not allowed. Possibilities include opening a CA
             which is already open, doing operation on CA which is not
             open, invalid command code for device type, or insufficient
             memory in the host processor to open the file.
10 - Simio - Bad file type value.
```

```
11 - Simio - Invalid row/column value for display.
12 - Simio - Invalid record length for device.
13 - Simio - Bad character in data for display (>= 0F0H)
14 - Simio - Simulated I/O shutdown by operator using the SIMIO softkey.
15 - Simio - Bad file name in rename command.
16 - Soft  - The characters in the file name string did not form a
             valid file name.
17 - Soft  - Maximum number of files (i.e. 20) exceeded.
18 - Soft  - Invalid FD(i.e. File Descriptor Number). Out of range or FD
             doesn't indicate an open CA buffer.
19 - Soft  - Either CA buffer for device not defined or no more disc
             CA buffers are available.
20 - Soft  - Attempt to "open" the printer or display for reading or
             read-write.
             Attempt to "open" the keyboard for writing or read-write.
             Attempt to "creat" the keyboard.
             Attempt to "read" or "write" a file that was not opened
             in the proper mode.
             Attempt to "write" in read-write mode and RW_WRITE_ENABLE
             is FALSE.
21 - Soft  - An attempt to read or write more than MAX_RECS (32767)
             records in a disc file.
```

Descriptions of Simulated I/O Library Routines

The following definitions are used in defining the routines which follow.

```
TYPE
   CA_BUFFER  = ARRAY[0..257] OF CHAR; {Maximum size CA buffer}
   CA_PTR     = ^CA_BUFFER;            {Pointer to CA buffer  }

   CSTRING    = ARRAY[0..49] OF CHAR;  {String to contain file name}
                                       {Data begins in 0th char.   }
                                       {and is terminated by first }
                                       {ASCII NUL character. CHR(0)}

   BYTEPTR    = ^BYTE;                 {Pointer to read or write buffer}
```

INIT_SIMIO_LIB

```
PROCEDURE INIT_SIMIO_LIB(DISPLAY_CA,  {display      CA buffer address}
                         PRINTER_CA,  {printer      CA buffer address}
                         RS232_CA,    {rs232        CA buffer address}
                         KEYBOARD_CA, {keyboard     CA buffer address}
                         DISC_1_CA,   {disc file #1 CA buffer address}
                         DISC_2_CA,   {disc file #2 CA buffer address}
```

```
        DISC_3_CA,    {disc file #3 CA buffer address}
        DISC_4_CA,    {disc file #4 CA buffer address}
        DISC_5_CA,    {disc file #5 CA buffer address}
        DISC_6_CA     {disc file #6 CA buffer address}

       : CA_PTR);
```

The programmer must explicitly call the procedure INIT__SIMIO__LIB before doing any simulated I/O operations. INIT__SIMIO__LIB initializes the data structures used by the Simulated I/O Library. The parameters are pointers to CA buffers, defined by the programmer, which are the interface to 64000 simulated I/O facility. The programmer must define a CA buffer for each device that will be used and for the maximum number of disc files that will be open at one time. If a CA buffer is not defined, then NIL should be the value for the corresponding parameter to INIT__SIMIO__LIB.

close

FUNCTION close(FD: SIGNED__16): SIGNED__16;

Parameters:

FD - must be the file descriptor number of an open file.

Return Value:

-1	- error detected.
0	- close successful.

The function close closes the file specified by the file descriptor FD. Any unwritten data in the CA buffer is first written. If other files are open to the same device, their operation is uneffected. After closing, no more operations may be performed using the specified file descriptor.

creat

FUNCTION creat(VAR NAME:CSTRING; PMODE:SIGNED__16): SIGNED__16;

Parameters:

NAME	- The name of the file to be created. Data characters in this array are terminated by the first ASCII NUL character (i.e. CHR(0)).

PMODE	- This parameter would contain a file protection mode in an I/O environment that supported them. It is ignored by the Simulated I/O Library.

Return Value:

-1	- error detected.
0 through 19	- **creat** successful. The returned value is a file descriptor number used for all subsequent references to the file.

The function **creat** creates a new file and opens it for writing. NAME contains the physical file name. For devices, this is the same as opening the file in write-only mode. For disc files, if the file already exists, it is deleted using the function **unlink**. A new empty disc file is then created. The function verifies the contents of NAME to insure that it contains a proper device or disc file name. Upon successful creation, the returned value is a file descriptor number which is used to identify the file for all subsequent operations.

open

```
FUNCTION  open(VAR NAME:STRING; MODE,PMODE:SIGNED_16): SIGNED_16;
```

Parameters:

NAME	- The name of the file to be opened. Data characters in this array are terminated by the first ASCII NUL character (i.e. CHR(0)).
MODE	- This value specifies the desired read-write mode. Zero specifies read-only, one specifies write-only, and two specifies read-write.
PMODE	- This parameter would contain a file protection mode in an I/O environment that supported them. It is ignored by the Simulated I/O Library.

Return Value:

-1	- error detected.
0 through 19	- **open** successful. The returned value is a file descriptor number used for all subsequent references to the file.

The function **open** opens an existing file in read-only, write-only, or read-write mode. NAME contains the physical file name. If the specified file does not exist, an error occurs. The function verifies the contents of NAME to insure that it contains a proper device or disc file name. Upon successful creation, the returned value is a file descriptor number which is used to identify the file for all subsequent operations.

overprint

FUNCTION overprint(FD:SIGNED__16): SIGNED__16;

Parameters:

FD	- must be the file descriptor number for a file open in the write-only or read-write mode.

Return Value:

-1	- error detected.
0	- **overprint** successful.

The function **overprint** is supposed to write a special control sequence to the file specified by FD. The control sequence would cause the next physical record to overstrike the present physical record when the file was printed on a line printer. The 64000 Simulated I/O facility does not support this capability. Instead, **overprint** simply writes a physical record boundary including any data already in the CA buffer.

page

FUNCTION page(FD:SIGNED__16): SIGNED__16;

Parameters:

FD	- must be the file descriptor number for a file open in the write-only or read-write mode.

Return Value:

-1	- error detected.
0	- **page** successful.

The function **page** writes a special control sequence to the file specified by FD. The control sequence causes a form feed operation when the file is printed on a line printer. If a CA buffer contains unwritten data, **page** writes a physical record boundary after the data in the CA. Then **page** writes a record containing an ASCII FORM FEED character (i.e. CHR(12)) followed by a SPACE character.

prompt

FUNCTION prompt(FD:SIGNED__16): SIGNED__16;

Parameters:

FD	- must be the file descriptor number for a file open in the write-only or read-write mode.

Return Value:

-1	- error detected.
0	- **prompt** successful.

The function **prompt** is supposed to write a special control sequence to the file specified by FD. The control sequence would cause any data in the CA buffer to be written to the file without writing a line marker (i.e record boundary). The 64000 Simulated I/O facility does not support this capability since writing data and writing a record boundary are inseparable operations. Instead, **prompt** simply writes a physical record boundary including any data in the CA buffer.

read

```
FUNCTION read(FD:SIGNED__16; PTR:BYTEPTR; N:SIGNED__16;
          VAR REC__BOUND:BOOLEAN): SIGNED__16;
```

Parameters:

FD	- must be the file descriptor number of a file open in the read-only or read-write mode.
PTR	- must be a pointer to the first byte of a buffer where data from the file will be placed.
N	- must be a value greater than 0 indicating the maximum number of bytes to be transferred. The buffer specified by PTR must be at least N bytes long.
REC__BOUND	- refers to a BOOLEAN variable that may be changed by read. REC__BOUND will be set to TRUE if **read** encounters the end of a physical record while transferring data. Otherwise, the value of REC__BOUND will be unchanged.

Return Value:

-1	- error encountered.
0	- If REC__BOUND is FALSE, then end-of-file detected. If REC__BOUND is TRUE, then a zero length record was encountered. In either case, no data was transferred.
1 through N	- number of bytes transferred from file to buffer.

The function **read** transfers data from the file specified by FD to a buffer whose 1st byte is specified by PTR. **read** will transfer data until N bytes are transferred or until the end of a physical record is encountered, whichever comes first. If a record boundary is encountered, REC__BOUND will be set to TRUE. Otherwise, REC__BOUND will be unchanged.

seek__rec

FUNCTION seek__rec(FD, RECNUM, RECSIZE, VARIABLE: SIGNED__16): SIGNED__16;

Parameters:

FD	- must be the file descriptor number of an open file.
RECNUM	- is a value indicating the desired record number. If RECNUM is less than or equal to zero, then the file will be positioned after the last existing record at the end-of-file mark. If RECNUM is greater than zero, it indicates the desired record number.
RECSIZE	- must be a number greater than 0 indicating the size, in bytes, of records in this file.
VARIABLE	- is a value indicating whether the records are fixed length or variable length. If VARIABLE is zero, then records are defined to fixed length with a length indicated by RECSIZE. If VARIABLE is non-zero, then records are defined to be variable length with a maximum length of RECSIZE.

Return Value:

-1	- error detected.
1 through RECNUM	- new file position where the first record in a file is number 1.

The function **seek__rec** positions the file specified by FD to either the end-of-file or to a specific record number. If a record number is specified and the file contains fewer records, then **seek__rec** positions to the end-of-file. If FD refers to the printer, keyboard, display, or rs232 devices, **seek__rec** does nothing and returns 1.

VARIABLE indicates whether records for this file are fixed length or variable length. If VARIABLE is zero, records are taken to be a fixed number of bytes specified by RECSIZE. If VARIABLE is non-zero, records are considered to be variable length with a maximum length specified by RECSIZE. That is, a record will consist of a physical record or RECSIZE bytes whichever is shorter. In either case, **seek__rec**, stops when it reaches the desired record (if specified) or end-of-file, whichever comes first.

NOTE: True random access is impossible with simulated I/O. All seeks are performed by first positioning the file to its beginning and then reading until the desired record is found. This implies very slow performance for **seek__rec**.

unlink

FUNCTION unlink(VAR NAME:CSTRING): SIGNED__16;

Parameters:

NAME	- The name of the file to be purged. Data characters in this array are terminated by the first ASCII NUL character (i.e. CHR(0)).

Return Value:

-1	- error detected.
0	- **unlink** successful.

The function **unlink** purges a file, whose physical file name is contained in NAME, from the 64000 file system. If NAME contains printer, display, keyboard, or rs232, no operation is performed and **unlink** returns zero. Otherwise, **unlink** attempts to purge a disc file. If the file does not exist, an error occurs. Otherwise the disc file is purged and put into the recoverable list.

write

FUNCTION write(FD:SIGNED__16; PTR:BYTEPTR; N, REC__BOUND: SIGNED__16):
SIGNED__16;

Parameters:

FD	- must be the file descriptor number of a file opened in the write-only or read-write mode.
PTR	- must be the address of the first byte of a buffer where the data to be written is found. If N is zero, PTR may be NIL.
N	- must be a value greater than or equal to zero indicating the number of bytes to be written.

REC__BOUND	- is a value indicating whether a physical record boundary should be written after the last data is transferred to the CA buffer. If REC__BOUND is zero, no physical record is written after the last byte of data is transferred to the CA buffer. If REC__BOUND is non-zero, then a physical record is written after the transfer.

Return Value:

-1	- error detected.
0 through N	- number of bytes tranferred from the buffer to the file. If the returned value is less than N, then an error occured.

The function **write** transfers data from a buffer whose first byte is specified by PTR to the file specified by FD. Data is first transferred from the buffer to the CA buffer. If the CA buffer is full and more data needs to be transferred, then a physical record is written to empty the CA buffer. After N bytes have been transferred, **write** checks REC__BOUND. If REC__BOUND is non-zero, a physical record is written containing all the data in the CA buffer.

NOTE: The end-of-file mark will always be written after any data written to a file. If **seek__rec** has been used to position to any place other than end-of-file, then performing a **write** operation will cause all data following that position to be irretrievably lost. Thus, true updating of records is impossible using simulated I/O.

In order to prevent this from happening accidently, the global variable RW__WRITE__ENABLE is given the default value FALSE. If one performs a write on a file in read-write mode and RW__WRITE__ENABLE is FALSE, an error will occur.

USER I/O ROUTINES

There are two sets of I/O routines: the standard procedures and functions for the Pascal/64000 I/O, and the simulated I/O routines which perform the Pascal/64000 I/O functions on the Software Development System. All Pascal/64000 I/O routines except **Perror** are microprocessor and target environment independent.

The simulated I/O routines run only on the 64000 emulation environment. Users are required to supply their own I/O routines for different running environments. The user's I/O routines should accept the same parameters and perform identically to the simulated I/O routines defined previously in this chapter. The following listing shows what simulated I/O routine(s) are called by a Pascal/64000 I/O function.

Pascal I/O Functions	Compiler generates calls to Pascal I/O Routines	Simulated I/O Routines called by the Pascal I/O Routines
F^	Pbufvar	read
APPEND	Pappend	open creat seek_rec close
CLOSE	Pclose	close unlink write
EOF	Peof	read
EOLN	Peoln	read
GET	Pget	read
LINEPOS	Plinepos	---
MAXPOS	Pmaxpos	---
OPEN	Popen	open creat close
OVERPRINT	Poverprint	overprint
PAGE	Ppage	page
POSITION	Pposition	---
PROMPT	Pprompt	prompt
PUT	Pput	write
READ	Pbufvar Pget Pread_char Pread_integer Pread_longreal Pread_real Pread_string Pread_unsigned	read read read read read read read read

Pascal I/O Functions	Compiler generates calls to Pascal I/O (PIOLIB)	Simulated I/O Routines called by the Pascal I/O Routines
READDIR	Pseek	seek_rec
	Pbufvar	read
	Pget	read
READLN	Preadln	read
RESET	Preset	open
		close
REWRITE	Prewrite	creat
		close
SEEK	Pseek	seek_rec
STRREAD	Pstringopen	---
	Pbufvar	read
	Pget	read
	Pread_char	read
	Pread_integer	read
	Pread_longreal	read
	Pread_real	read
	Pread_string	read
	Pread_unsigned	read
STRWRITE	Pstringopen	---
	Pbufvar	read
	Pput	write
	Pwrite_char	write
	Pwrite_integer	write
	Pwrite_longreal	write
	Pwrite_real	write
	Pwrite_string	write
	Pwrite_unsigned	write
WRITE	Pbufvar	read
	Pput	write
	Pwrite_char	write
	Pwrite_integer	write
	Pwrite_longreal	write
	Pwrite_real	write
	Pwrite_string	write
	Pwrite_unsigned	write
WRITEDIR	Pseek	seek_rec
	Pbufvar	read
	Pput	write
WRITELN	Pwriteln	write

A file can be accessed only if it is open, and it should be closed by the user explicity. Therefore, the I/O routines **open, creat,** and **close** are essential and must be supplied by the user before any I/O can be called. In addition to these basic routines, the routine **read** is needed if a call to **GET, EOF, EOLN, READ,** or **READLN** is called in the user's Pascal program. For unwanted I/O functions, such as OVERPRINT, users can implement a dummy function which always returns -1 to the calling routine. Any call to this function will result in an I/O error or warning, depending on the implementation of **Perror.**

The Pascal I/O routine **Perror** is microprocessor and environment dependent since it terminates program execution. Users are required to supply their own **Perror** routine to handle I/O errors for their specific target environment.

The compiler option **$IOCHECK$** allows the user to select whether I/O errors are handled by the "system" or the user's program. All I/O errors cause **Perror** to be called. If **IOCHECK** is **OFF**, then **Perror** simply stores the parameter **ERRCODE** into the global variable **IOR** and returns. If **IOCHECK** is **ON**, then **Perror** performs an error handling procedure which is appropriate for the target environment. For emulation, this is termination of the program.

Perror functions as follows:

```
PROCEDURE Perror(ERRCODE : SIGNED_8)

BEGIN
IOR := ERRCODE;
IF Piocheck THEN  {Piocheck has the value of the compiler }
   BEGIN          {directive IOCHECK.                     }
       { run-time error }
       { handler        }
   END;
END; { Perror }
```

NOTES

Chapter 7

STANDARD PROCEDURES AND FUNCTIONS

DYNAMIC ALLOCATION/DE-ALLOCATION PROCEDURES

General Information

Pascal/64000 allows variables to be created during program execution. The space, called the "heap", allocated to dynamic variables can then be de-allocated and later re-allocated to another variable. Dynamic allocation and de-allocation are useful when variables are needed only temporarily, and when a program contains data structures whose maximum size may vary each time the program is run. Examples are temporary buffer areas and dynamic structures such as linked lists or trees. Dynamic variables are not explicitly declared and cannot be referred to directly by identifiers.

The standard procedure **NEW** is used to create variables. The standard procedure **DISPOSE** is used to deallocate variables.

When it is known in advance that a group of dynamic variables may be needed on a short term basis, the state of the heap, before the short term variables are allocated, can be recorded by using the predefined procedure **MARK.** When the short term variables are no longer needed, the heap can be returned to the original condition by using the predefined procedure **RELEASE.** All variables allocated after the procedure **MARK** are removed.

An attempt to allocate variables that require more space than available in the heap will cause an error message and aborting of the program.

The following paragraphs describe in greater detail the procedures **NEW, DISPOSE, MARK,** and **RELEASE.**

NOTE

The routines **NEW**, **DISPOSE**, **MARK**, **RELEASE**, and **INTIHEAP** are state and should be defined with **$RECURSIVE OFF$.** **INITHEAP** is described in the compiler supplement for your processor, when applicable.

NEW(p)

The procedure **NEW** is used to allocate memory space for a dynamic variable. (p) is a variable of type **POINTER**. (p) can only point to a variable of a particular type *T*, and therefore is said to be bound to *T*.

When the procedure **NEW(p)** is called, a section of the heap large enough for a variable of type *T* is allocated and the address of that space is held in pointer (p).

If *T* is a record with variants, then the amount of space allocated is the amount required by the fixed part of the record, plus the amount required by the largest variant.

DISPOSE(p)

DISPOSE(p) indicates that the dynamic variable p^ is no longer needed. The value of the pointer p is set to **NIL**. The space occupied by p^ is returned to the heap.

MARK(p)

MARK(p) is a predefined procedure having one parameter, a pointer variable, that records the **HEAP** state at the time **MARK** is executed. Calling **MARK(p)** causes assignment of the first free address in the **HEAP** to (p). The value of (p) may not change between **MARK** and **RELEASE.** Any execution of the procedure **NEW** will build new data structures, starting with the address held in (p).

RELEASE(p)

RELEASE is a predefined procedure having one parameter, a pointer variable, that restores the **HEAP** to the state present at the time of **MARK(p)**. The value of (p) may not change between **MARK** and **RELEASE.** All dynamic variables created after **MARK** are effectively destroyed, and the memory space occupied by those variables is available for allocation to new dynamic variables. Be sure that no pointer variables point to dynamic structures created after the MARK procedure.

ARITHMETIC FUNCTIONS

There are seven predefined arithmetic functions in Pascal/64000. Each of these functions is passed in an arithmetic expression as a parameter and returns a numeric value.

The type returned depends on the type of parameter passed. ABS returns an INTEGER if an INTEGER value is passed. The other functions return REAL if an INTEGER is passed. All the functions return REAL if REAL is passed and return LONGREAL if LONGREAL is passed.

To compute the values of the functions, Pascal/64000 uses system routines and compiler-defined algorithms as follows:

ABS

ABS(X)	-	computes absolute value of X.

need 4
SQRT

SQRT(X)	-	Computes the square root of X. If X < 0 then a run-time error occurs.

EXP

EXP(X) -	Computes e (base of the natural logarithms) to the power of X.

LN

LN(X) -	Computes the natural logarithm of X. If $X < 0$ then a run-time error occurs.

SIN, COS

SIN(X), COS(X) -	Computes the sine and cosine of X, where X is in radians.

ARCTAN

ARCTAN(X) -	Computes the arctangent of X in radians.

PREDICATES

The following procedure returns a Boolean result.

ODD

ODD(X) -	The procedure **ODD** returns **TRUE** if the value of the integer expression X is odd, **FALSE** otherwise.

TRANSFER FUNCTIONS

TRUNC

TRUNC(X) - The function **TRUNC** returns an integer result that is the integral part of the real or longreal expression X. The absolute value of the result is not greater than the absolute value of X. An error will occur if the result is not within the integer range.

Examples: TRUNC(5.61) returns 5
TRUNC(-3.38) returns -3
TRUNC(18.999) returns 18

ROUND

ROUND(X) - The function **ROUND** returns the integer value of the real or longreal expression X rounded to the nearest integer. If X is positive or zero, then **ROUND**(X) is equivalent to **TRUNC**(X + 0.5); otherwise **ROUND**(X) is equivalent to **TRUNC**(X - 0.5). An error will occur if the result is not in the integer range.

Examples:

```
ROUND(3.1)      returns   3
ROUND(-6.4)     returns  -6
ROUND(-4.6)     returns  -5
```

ORDINAL FUNCTIONS

The ordinal functions are: ORD, CHR, SUCC, and PRED.

ORD

ORD(X) - where X is an expression of ordinal type. The function **ORD** returns the ordinal number associated with the value of X. The type of the result is INTEGER. If the parameter is compatible with INTEGER, then the parameter value is returned as the result. If X is of type **CHAR**, then the result is an integer value between 0 and 255, determined by the ASCII ordering. If X is of any other ordinal type (i.e., a predefined or user-defined enumeration type), then the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers starting at zero.

The **ORD** value of -1 is -1; the **ORD** value of 1000 is 1000. The **ORD** value of 'a' is 97, the **ORD** value of 'A' is 65.

The predefined type **BOOLEAN**, for example, is defined:

TYPE BOOLEAN = (FALSE, TRUE) and therefore **ORD**(FALSE) returns 0, and **ORD**(TRUE) returns 1.

The same method is used to detertmine the ordinality of an element in a user-defined enumeration type. For example, given the declaration:

TYPE color = (red, blue, yellow); the **ORD**(red) returns 0, **ORD**(blue) returns 1, and **ORD**(yellow) returns 2.

CHR

CHR(X) - where X is an integer expression. The function **CHR** returns the character value whose ordinal number is equal to the value of the integer expression X. If X is greater than 255, then only the least significant 8 bits of X is used to form the character. For any character ch, the following is true: CHR(ORD (ch)) = ch.

Examples:

The value of 63 returns the CHR '?', the value 100 returns the CHR 'd', the value 13 returns the CHR 'carriage return', the value 75 returns the CHR 'K'.

SUCC

SUCC(X) - where X is an expression of ordinal type. The function **SUCC** returns a result having an ordinal one greater than the expression X. The result is of a type identical to that of X. If no such value exists, no error is reported at the function call, but a run-time error will occur if the value is assigned to a variable of the ordinal type. Given the declaration: TYPE color = (red, blue, yellow); **SUCC**(red) returns blue, and **SUCC**(yellow) returns a value that is not of type COLOR. If X is 1, then **SUCC**(X) is 2. If X is -5, then **SUCC**(X) is -4. If X is 'a', then **SUCC**(X) is 'b'. If X is FALSE, then **SUCC**(X) is TRUE.

PRED

PRED(X) - where X is an expression of ordinal type. The function **PRED** returns a value having an ordinal value one less than X. If no such value exists, no error is reported at the function call, but a run-time error will occur if the value is assigned to a variable of the ordinal type. Given the declaration:

TYPE day = (monday, tuesday, wednesday); the following is true: **PRED**(tuesday) = monday, and **PRED**(monday) returns a value that is not of type day.

Examples:

The **PRED**(1) is 0, the **PRED**(-5) is -6, the **PRED**('b') is 'a', the **PRED**(TRUE) is FALSE.

ADDR FUNCTION

ADDR

ADDR(V) - returns the address of any variable V. The result is a pointer that is compatible with any pointer type.

Example:

```
VAR A : ARRAY[1..10] OF INTEGER;
    P : ^INTEGER;
BEGIN
P := ADDR(A[5]);
```

SHIFT AND ROTATE FUNCTIONS

SHIFT

SHIFT (E,N) - E is an expression of an ordinal type and N is an expression that is compatible with INTEGER. If N is positive then the binary value of E is shifted left N times. If N is negative then the binary value of E is shifted right N times. The type returned is identical to the type of E.

Example:

```
VAR I: INTEGER;
BEGIN  I := 4;
       I := SHIFT(I,1);  {I now contains 8}
```

ROTATE

ROTATE(E,N) - E is an expression of an ordinal type and N is an expression compatible with INTEGER. If N is positive, then the binary value of E is rotated to the left N times. If N is negative then the binary value of E is rotated to the right N times. The type returned is identical to the type of E.

Example:

```
VAR I : SIGNED__16;
BEGIN  I := 8000H;
       I := ROTATE(I,1); {I now contains 1}
```

Chapter 8

COMPILER OPTIONS

INTRODUCTION

The compiler interprets the following construct as a compiler directive:

$<compiler__options>$

Compiler options may be inserted between any two tokens (identifiers, numbers, string literals, and special symbols). They are used to inform the compiler about changing needs within the program. A compiler option is a separator (as is a space or a comment) in the Pascal program. Compiler options must begin with a dollar sign and close with a dollar sign. A compiler error will result if no closing dollar sign appears on the line. A compiler option must exist entirely on one line.

The compiler option specification may include an option value. If no option value is specified, then, for options that require an ON - OFF value, **ON** is assumed. Otherwise, for options that require an integer or string literal value, the option is set to its default value.

TRUE and plus sign (+) are equivalent to **ON**. FALSE and minus sign (-) are equivalent to **OFF**.

The compiler option syntax is defined here, and the syntax diagram is shown in figure 8-1.

```
<compiler options> ::= <option> {<separator> <option>}

<separator>      ::= ; | , ;

<option>        ::= <identifier> <option value> | <empty>

<option value>   ::= ON | OFF | <signed integer> | <string literal>| <empty> |
                 ::=  + |  -  | TRUE | FALSE
```

All directives, prior to use in a source program, will assume their initial value when the compiler is called.

AMNESIA [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes the compiler to forget the contents of registers after the registers are used in an operation. This directive may be used to ensure that variables representing memory mapped I/O ports are reloaded everytime they are needed.

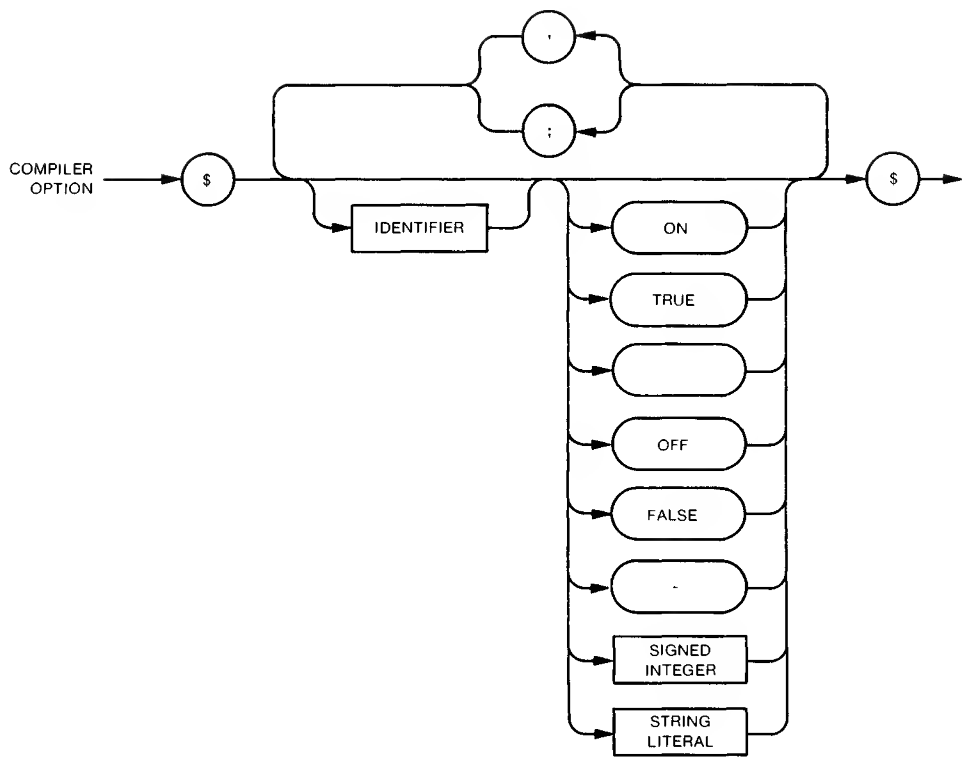


Figure 8-1. Compiler Options Syntax

ANSI [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes a warning message to be issued for any feature of Pascal/64000 which is not part of "standard" Pascal. This feature is useful for identifying areas that must be considered for program transportability.

ASM__FILE

Initialized Value: **OFF**

Description:

The source file is created into a file whose name consists of the letters "ASM" followed by the microprocessor designator (e.g. ASM8085, ASMZ80). This assembler source will be accepted by the assembler as a source file for the selected microprocessor. If the LIST__CODE directive is also **ON** when ASM__FILE is **ON**, the assembler source file will also contain intermixed Pascal source lines as assembler comments.

ASMB__SYM [ON] [OFF]

Initialized Value: **ON**

Description:

ON causes the compiler to generate an asmb__sym file for use during emulation. **OFF** suppresses the generation of the file.

DEBUG [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes all arithmetic operations to be checked for overflow, underflow, or divide by zero operation. (See specific Pascal/64000 microprocessor-dependent supplement for run-time error descriptions.)

EMIT__CODE [ON] [OFF]

Initialized Value: **ON**

Description:

ON specifies that executable code is to be emitted to the relocatable code file.

END__ORG

Description:

Used to change variable address assignment from absolute to relocatable mode.

EXTENSIONS [ON] [OFF]

Initialized Value: **OFF**

Description:

ON allows the programmer to use the microprocessor oriented extensions to the Pascal language. **OFF** disallows the use of these language extensions. The extensions include functional type changing, the address functions, the BYTE data type, built-in functions, **SHIFT** and **ROTATE**, and non-decimal constant representations.

EXTENSIONS **ON** turns RECURSIVE **OFF** and EXTENSIONS **OFF** turns RECURSIVE **ON**.

EXTVAR [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes all variables defined to be declared EXTERNAL until a subsequent EXTVAR **OFF** is encountered. No local storage is allocated in this module for such variables. Symbols used in one program module but defined in another, must be declared as external variables. Externals must have been declared to be global in another program.

FULL__LIST [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes INCLUDE files to be listed and MACROS to be expanded in the listfile. Lines with errors will be shown whether this directive is **ON** or **OFF**.

GLOBPROC [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes all main block procedures defined to be declared global until a subsequent GLOBPROC **OFF** is encountered. This allows access by other modules. Global procedures can be called from outside the program. When called from outside the program, they must be declared external in the calling module.

GLOBVAR [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes all main block variables defined to be declared global until a subsequent GLOBVAR **OFF** is encountered. This allows access to these variables by other modules where the variables have been declared to be external.

IOCHECK [ON] [OFF]

initialized Value: **ON**

Description:

ON causes all input/output procedures and functions to terminate program execution when an I/O error is detected. **OFF** causes input/output procedures and functions, when an I/O error is detected, to store an error code in a global variable and return. In this mode, a program can recover from I/O errors.

LINE__NUMBERS [ON] [OFF]

Initialized Value: **ON**

Description:

ON causes the compiler to generate symbols for the Pascal/64000 source line numbers. These symbols are found in the asmb__sym file after the compilation. They may be used during emulation to trace the execution of a Pascal/64000 program by source line number. The symbols are constructed by placing a pound sign (#) in front of the line number. Line number symbols are created only for lines that cause executable code to be generated (i.e. line number symbols will not be created for lines in the *TYPE* and *VAR* sections of the program).

LIST [ON] [OFF]

Initialized Value: **ON**

Description:

ON causes the source file to be copied to the listfile. **OFF** suppresses the listing except for lines that contain errors.

LIST__CODE [ON] [OFF]

Initialized Value: **OFF**

Description:

ON specifies that the program listfile will contain the symbolic form (assembly language) of the code produced, intermixed with the source lines.

LIST__OBJ [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes the listing to contain the relocatable object code generated by the third pass of the compiler.

OPTIMIZE [ON] [OFF]

Initialized Value: **OFF**

Description:

ON may cause certain run-time checks to be ignored, such as pre-checking the range values of a CASE statment. This mode is typically susceptible to bad out-of-range data at run time. The directive should only be used for well-structured programs that have been thoroughly debugged. Refer to the specific microprocessor-dependent supplement for additional information.

ORG number

Description:

All variables declared until END__ORG is encountered will be allocated sequential absolute addresses starting from "number". "number" may be represented with a hexadecimal constant.

The use of this compiler directive to assign variables to absolute memory locations does not allocate any absolute memory space. The reference to these variables are explicit absolute addresses in the relocatable file. The linker will not detect or report a memory overlap if the user's absolute addresses interfere with other defined memory areas.

PAGE

Initialized Value: **null**

Description:

This option causes a form feed to be output to the listfile.

RANGE [ON] [OFF]

Initialized Value: **OFF**

Description:

ON causes the compiler to generate code to check array indices, value parameters, variable set elements, and subrange assignments for legal values.

RECURSIVE [ON] [OFF]

Initialized Value: **ON**

Description:

ON causes all procedures, declared to be compiled, to allow recursive or reentrant calling sequences until a subsequent RECURSIVE **OFF** is encounterd. **OFF** causes procedures to be compiled in a static mode which does not allow for recursive or reentrant calling sequences.

SEPARATE [ON] [OFF]

Initialized Value: **OFF**

Description:

ON enables the separation of program and constants and data such that program code and constants are put in the PROG relocatable area and data in the DATA relocatable area. **OFF** puts program code, constants, and data into the PROG relocatable area. Refer to the specific microprocessor-dependent supplement for additional information.

TITLE "string"

Initialized Value: **null**

Description:

The first 50 characters of the string are moved into the header line printed at the top of each subsequent page or the listfile.

USER__DEFINED

Initialized Value: **null**

Description:

Pascal/64000 allows the user to redefine the semantics of certain operators in the language. User defined operators are created by using the option $USER__DEFINED$ during the declaration of a type in the **TYPE** section. For user defined operators, the compiler will not generate in-line code to perform the operations, but the compiler will generate calls to user provided run-time routines. The run-time routine name will be a composite of the user's type name and the operation being performed, TYPENAME__OPERATION. The first eleven characters of the user's type name are concatenated with an underscore and three characters identifying the operation. Following is a list of the operators that can be user defined and the run-time routine names that the compiler will create when the operations are used.

Operation	**Symbol**	**Run-time Routine**
1) Add	+	<typename>__ADD
2) Negate	-	<typename>__NEG
3) Subtract	-	<typename>__SUB
4) Multiply	*	<typename>__MUL
5) Divide	/ or DIV	<typename>__DIV
6) Modulus	MOD	<typename>__MOD
7) Equal Comparison	=	<typename>__EQU
8) Not Equal Comparison	<>	<typename>__NEQ
9) Less Than or Equal to Comparison	<=	<typename>__LEQ
10) Greater Than or Equal to Comparison	>=	<typename>__GEQ
11) Less Than Comparison	<	<typename>__LES
12) Greater Than Comparison	>	<typename>__GTR

Refer to the specified microprocessor-dependent supplement for additional information on this directive.

WARN [ON] [OFF]

Intialized Value: **ON**

Description:

Specifies that warning messages be written to the listing file. When this directive is **OFF**, only error messages will be displayed and listed.

WIDTH number

Initialized Value: **240**

Description:

The number specifies the number of significant characters (width) in the source file to be compiled. Additional characters are ignored and if WARN is **ON**, a warning message will be generated.

THE PREPROCESSOR PASS

A special preprocessor pass is available with the Pascal/64000 compiler. This preprocessor allows the user to use include files, macros, and conditional compilation. To make use of the preprocessor, a special compiler directive must be used on the first line of the source program. In addition to the processor name, the word PREPROCESS must be present as follows:

```
"Z8001" PREPROCESS
```

The word PREPROCESS must follow immediately after the processor name. Anything other than PREPROCESS will be ignored and the preprocessor will not be invoked.

General Syntax

All preprocessor instructions must have a pound sign (#) in column 1 of the line. This should be followed by the appropriate instruction. The preprocessor is line oriented. Preprocessor instructions may appear anywhere in the source file and do not affect the general syntax of the program. Preprocessor instructions are not affected by comments.

INCLUDE FILES

#INCLUDE <FILE>
or
#INCLUDE "<FILE>"

This instruction causes the appropriate file to be included into the source. Anything on the line after the file name will be ignored. Both forms of the instruction are equivalent.

Examples:

```
#INCLUDE MYFILE
#INCLUDE "MYFILE:MYUSER"
#INCLUDE MYFILE:MYUSER:3
```

MACROS

#DEFINE <MACRO__NAME> <TEXT>

This instruction will cause all subsequent occurrences of <MACRO__NAME> to be replaced by <TEXT>. <MACRO__NAME> may be any upper-case or lower-case identifier. <TEXT> may be any text.

```
#DEFINE <MACRO_NAME>(<FORMAL_PARAMETER>,...,<FORMAL_PARAMETER>)<TEXT>
```

There must not be any spaces between the macro name and the (.

This form of the instruction is similar to the previous form except that parameters may be passed. Each occurrence of the formal parameters in the text is replaced by the parameters being passed. To refer to a macro of this form, the macro name should be used, followed by the parameters separated by commas and within parentheses. Parameters may themselves contain parentheses, and commas within inner parentheses do not separate parameters. Parentheses and commas within strings will be ignored.

Both of the above forms of macros will not be expanded within strings. They will be expanded within comments. Normally a macro declaration is limited to one line. However, it may be expanded to the next line by ending a line with a backslash (\). The backslash will not be part of the expanded text. Macros may refer to other macros, and are independent of normal scoping. A macro may be redefined by declaring it again. If this is done, a warning will be given.

Example:

In a statistics program, one might write:

```
        #DEFINE SIGMA(X) SUM := SUM + X;\
        SUMSQUARES := SUMSQUARES + (X) * (X);\
        NUMBER := NUMBER + 1;
```

Later in the code the statement SIGMA(VALUE) will be replaced by:

```
        SUM := SUM + VALUE;
        SUMSQUARES := SUMSQUARES + (VALUE) * (VALUE);
        NUMBER := NUMBER + 1;
```

Note the parentheses around the X's in the second line. If this was not done, SIGMA(A+B) would cause the second line to be:

```
                    SUMSQUARES := SUMSQUARES := SUMSQUARES + A+B * A+B;
```

#UNDEF <MACRO__NAME>

This instruction will undefine the previously defined macro.

Conditional Compilation

#IF <EXPRESSION>

This instruction will cause the following code to be compiled only if <EXPRESSION> evaluates to TRUE. <EXPRESSION> may be any expression involving:

a. Integers.

b. Integer operators +, -, *, DIV, and MOD.

c. Boolean operators AND, OR, and NOT.

d. Relational operators =, <>, <, >, <=, and >=.

The result of the expression must be a Boolean value. The conditional compilation will continue until

```
                        #ENDIF
```

is encountered. The instruction:

```
                        #ELSE
```

will turn **ON** compilation if it is off, or turn it **OFF** if it is on.

#IFs may be nested. Within a #IF FALSE, nested #IFs will not have any effect on compilation.

Since the preprocessor is not affected by normal compiler syntax, constants defined in the **CONST** section of the program can not be used in the #IF instruction. However, it is possible to define a macro which will be usable in the #IF statement. For example, if the user has two systems he may write:

```
#DEFINE SYSTEM__1 TRUE
  ...
#IF SYSTEM__1
  ...
#ELSE
  ...
#ENDIF
```

When compiling for SYSTEM 2, change define SYSTEM__1 as being FALSE.

For three or more systems the following may be used:

```
#DEFINE SYSTEM 1
  ...
#IF SYSTEM = 1
  ...
#ELSE
#  IF SYSTEM = 2
  ...
#  ELSE
  ...
#  ENDIF
#ENDIF
```

Two other forms of the #IF instructions are available:

```
#IFDEF <MACRO__NAME>
```

will be true if the macro is defined.

```
#IFNDEF <MACRO__NAME>
```

will be true if the macro is not defined.

Compiler Directives

Normally, compiler directives are treated like any other text by the preprocessor and do not affect operation of the preprocessor. If it is desired for a directive to affect the preprocessor, a pound sign (#) must appear in column one and should be followed by the dollar sign ($) to indicate the directive. The directive will affect both the preprocessor and pass 1. The line containing the directive will not be expanded. Although any valid directive may appear on this line, only WIDTH, WARN and ANSI, will affect the preprocessor.

Examples:

```
#$WIDTH = 80$
#$WARN ON, ANSI OFF$
```

COMPILER GENERATED SYMBOLS

Compiler Generated Labels

Whenever the symbol "proc" appears it refers to the name of the enclosing procedure (or main program) truncated if necessary so that the total label will fit into 15 characters.

PROCEDURE ENTRY. The procedure entry has the label **proc**, i.e., the procedure name itself. This label will be declared global if the procedure is global. (The main program is always global.)

END LABEL. The end of a procedure is indicated by the label **Eproc**. This label marks the end of the PROG section associated with the procedure. This includes any data associated with the procedure which is in the PROG section (due to the value of the SEPARATE option). The end label will be declared global if the procedure is global. This label may be used in a trace as in trace only address range **proc thru Eproc**.

RETURN LABEL. The return instruction from a procedure is always labelled **Rproc**. This label will be declared global if the procedure is global.

DATA LABEL. If a procedure has an associated data area in memory, the data area will be marked **Dproc**. The data label is never global. It may be used in tracing local data as in trace address **Dproc+N** where N may be calculated from the relocation information in the listing.

USER LABELS. When a numbered label is used in Pascal the generated label is of the form **LABEL__NN** where NN is the number of the label. These labels are always local.

JUMP LABELS. These are labels generated by the compiler jumps from statements such as IF, FOR, WHILE, CASE, REPEAT, etc. The labels are of the form **procLNN__XXXX** where proc is truncated to seven characters, NN is a unique number based on the procedure and XXXX is a unique number for the labels. Jump labels are always local and will normally be unique within a program.

Certain processors may make use of other types of labels. See specific processor supplement manuals for details.

DUPLICATE SYMBOLS

Although labels aid in program tracing, they generate a potential for duplicate symbols. If these symbols are local, they will not cause a problem unless the ASM__FILE is assembled, or an attempt is made to trace on one of these variables. If the symbols are global, an error will occur at link time. The following can cause duplicate symbols.

If the first 14 characters of two procedure names match, the D, E, and R labels will be duplicated. If the procedure proc exists, as well as a user symbol such as Eproc (any procedure or global variable), a duplicate symbol will occur. Using the same procedure name twice, although legal due to nesting, will cause local duplicate symbols.

Similarly, using the same LABEL number in two procedures will cause duplicate local symbols. Using a reserved assembler symbol (such as a register name) may cause duplicate symbol errors in the ASM_FILE.

In the following example, note the following:

- The variables A, B, and C can be accessed as DTEST, DTEST+4, and DTEST+8.
- The procedure HAS_A_LONG_NAME was trucated to form the other labels. However, emulation also truncates identifiers to 15 characters (i.e., DHAS_A_LONG_NAME and DHAS_A_LONG_NAM are equivalent).
- The variable X can be accessed as DHAS_A_LONG_NAM.
- The use of two LABEL 5 statements, although valid in this sample Pascal program, will cause the symbol, LABEL_5, to be generated twice. This will result in a duplicate symbol error if the ASM_FILE is assembled, and one of the symbols cannot be traced.

Example:

```
 1 0000  1   "Z8002"
 2 0000  1   $RECURSIVE OFF$
 3 0000  1   PROGRAM TEST;
 4 0000  1   LABEL 5;
 5 0000  1   VAR
 6 0000  1      A:INTEGER;
 7 0004  1      B:INTEGER;
 8 0008  1      C:INTEGER;
 9 000C  1
10 000C  1      PROCEDURE HAS_A_LONG_NAME;
11 0000  2      LABEL 5;
12 0000  2      VAR X:INTEGER;
13 0004  2      BEGIN
      0000           HAS_A_LONG_NAME
14 0000  2         IF A=B THEN X := 5
      0000                 LDL   RR12,DTEST
      0004                 CPL   RR12,DTEST+00004H
      0008                 JP    NE,HAS_A_L01_1
15 000C  2         ELSE X := 3;
      000C                 LDL   RR12,#000000005H
      0012                 LDL   DHAS_A_LONG_NAM,RR12
      0016                 JP    ,HAS_A_L01_1
      001A           HAS_A_L01_0
      001A                 LDL   RR12,#000000003H
      0020                 LDL   DHAS_A_LONG_NAM,RR12
      0024           HAS_A_L01_1
16 0024  2         5:
17 0024  2      END;
```

```
      0024           LABEL_5
      0024           RHAS_A_LONG_NAM
      0024               RET
      0026           DHAS_A_LONG_NAM
      0026               RMB     00004H
18 0000   1
19 0000   1  BEGIN
      002A           EHAS_A_LONG_NAM EQU $-1
      002A           TEST
      002A               LDA     R15,STACK-
20 002E   1  5:
21 002E   1  END.
      002E          LABEL_5
      002E          RTEST
      002E              GLOBAL    RTEST
      002E              JP        ,Zendprogram
      0032          DTEST
      0032              RMB       0000CH
      0032          ETEST              EQU $-1
      0032
      0032              GLOBAL    ETEST
      0032
                        GLOBAL    TEST
                        EXTERNAL  STACK_ EXTERNAL Zendprogram
                        END       TEST
```

Chapter 9

HOW TO COMPILE A PROGRAM

GENERAL

Pascal/64000 uses a three-pass compilation process. The first pass, which is machine independent, reads the Pascal source, checks for lexical, syntax, and semantic errors, and produces an intermediate language file on disc. The second pass reads the intermediate language file and generates code for the chosen microprocessor by producing a tokenized assembler file on disc. The third pass reads the tokenized assembler file, generates a relocatable object file if there were no errors in the first two passes, and generates a list file if requested.

The optional list file may contain source lines only or source lines mixed with the generated assembly language code if requested.

COMMAND SYNTAX

The following pages provide the formal syntax definition for the compile command.

compile

SYNTAX

```
                           ┌              ┌          ┐ ┐
                           │              │ <FILE>   │ │
                           │              │ display  │ │
compile  <FILE>            │   listfile  {           } │
                           │              │ printer  │ │
                           │              │ null     │ │
                           └              └          ┘ ┘
                  ┌         ┌        ┐                                          ┐
                  │         │ list   │                                          │
                  │ options │        │ [expand][nocode][xref][comp_sym]         │
                  │         │ nolist │                                          │
                  └         └        ┘                                          ┘
```

Default Values

listfile	The default is to the predefined listfile. If there is no predefined list file, a null list file is the default.
options	If no entry is made for any of the options, the default values will be:

list/nolist	LIST directives in the source file will be obeyed.
expand	LIST_CODE and FULL_LIST directives in the source file will be obeyed.
nocode	EMIT_CODE directives in the source file will be obeyed.
xref	Default OFF - no symbol cross-reference listing generated.
comp_sym	Default OFF - no symbol file created.

compile (Cont'd)

FUNCTION

The compile command tells the compiler to translate a Pascal source program (file) into relocatable object code for a microprocessor.

Command Parameters:

<FILE>	A variable representing the source file name, userid, and disc number. The syntax for <FILE> is:

<FILE> => <FILE NAME> [:<USERID>][:<DISC#>]

where:

<FILE NAME>	- Up to nine alphanumeric characters, beginning with an upper-case alphabetic character.
<USERID>	- Up to six alphanumeric characters, beginning with an upper-case alphabetic character.
<DISC#>	- Represents the logical unit number of the system disc on which the source file is located. Allowable entries are decimal numbers representing the desired disc number.

The file type must be a source file; no other file type can be specified with the compile command. The first line of the source file must be the name of the target processor, enclosed in quotation marks (e.g. - "8085").

listfile	A key word which specifies a listing file for compiler output. When listfile is specified, one of the following must be specified also:

<FILE> display printer null

compile (Cont'd)

options — A key word which allows specification of options for the compile process. When "options" is specified, one or more of the following may be specified:

list
nolist
expand
nocode
xref
comp_sym

where:

list or nolist - allows specification of the source program list with error messages or no source listing except for error messages. All LIST directives in the source file are ignored.

expand - specifies a list of all source lines with an expansion of the assembly language. Also shows INCLUDE files and expanded MACROS if used. All LIST_CODE and FULL_LIST directives in the source file are ignored.

nocode - specifies the suppression of object code generation. Only the source code will be listed in pass 2.

xref - specifies a symbol cross-reference listing for the source file.

comp_sym - this file is created by the compiler when requested and contains the entire compiler symbol table for use in creating **comp_db** by the linker (refer to chapter 10 for a description of **comp_db**).

HOW TO COMPILE A PROGRAM

The usual process of software generation with the compiler is as follows:

a. Create source program files with editor.

b. Compile source program files.

c. Link relocatable files.

d. Emulate absolute files.

e. Debug.

The following sections of this manual will provide insight into the structure of the source file, compiling the source file, and linking relocatable files. Refer to the appropriate microprocessor-dependent supplement for information on emulating and debugging.

THE SOURCE FILE

The Pascal/64000 compiler takes as input a program source file created with the editor. The basic form of a source file is:

```
"8085"
PROGRAM Name;
   .
   .            {comments}
   .
CONST
  ...;
TYPE
  ...;
VAR
  ...;
PROCEDURE Procedure_name (Parameter1 : Type);
  .
  .
 BEGIN
   .
   .
 END;
BEGIN
   .
   .      {main program code}
   .
END.
```

The first line of the source file must be the special compiler directive which indicates the processor for which the file will be compiled. In the example form given above, the 8085 microprocessor is specified.

All key words in the source program must be upper-case, but identifiers may be lower case. When use of the 64200 emulator is planned, the global identifiers must begin with an upper-case letter if the user wishes to access these names symbolically during emulation. (In emulation, only emulation command keywords may start with a lower case letter. All user symbols must start with an upper case letter.)

COMPILING

When your program is complete, it is ready for compiling. To compile a program, press the (compile) softkey. The key word, *compile*, will appear on the command line and the softkey configuration will change to:

```
(<FILE>) (    ) (    ) (    ) (    ) (    ) (    ) (    )
```

Next, enter the source file you want to compile. When the file has been entered, the softkey configuration will change to:

```
(    ) (listfile) (options) (    ) (    ) (    ) (    ) (    )
```

If you want a listing file for the compile program, press the (listfile) softkey. The key word, *listfile*, will appear on the command line and the softkey configuration will change to:

```
(<FILE>) (display) (printer) ( null ) (    ) (    ) (    ) (    )
```

At this point, choose the listing file you want as indicated by the softkeys. If you do not choose a listfile, the compiler will default to the predefined listfile that was chosen when the userid was set. (Refer to the System Software manual for setting the userid.)

You may now select the appropriate compile options.

If you do not want to specify any options, press the (RETURN) key to compile your source file.

If you want to specify options, press the (options) softkey. The key word, *options*, will appear on the command line and the softkey configuration will change to:

```
(list) (nolist) (expand) (nocode) (xref) (comp_sym) (____) (____)
```

Press the softkey of the option or options you want to specify; then press the (RETURN) key to compile your source file.

OUTPUT LISTINGS

The compiler will output relocatable code and make listings according to the options specified or their default value. The following examples show typical output listings that the compiler will produce.

The following listing is an example of an 8085 output listing to the printer with a cross reference (xref) table.

```
FILE: ERROR:P              HP Pascal/64000[A.1]

   1  0000   1     "8085"
   2  0000   1     PROGRAM FACTORIAL;
   3  0000   1       VAR
   4  0000   1           I: INTEGER; {Loop control variable}
   5  0002   1           N: INTEGER;
   6  0004   1           FACT: INTEGER;
   7  0006   1      BEGIN
   8  0000   1       FACT:= 1;
   9  0000   1       FOR I:= 1 TO X DO  {Deliberate error}
****  ERROR    ??               ^104^103
  10  0000   1        FACT:= FACT*I;
  11  0000   1      END.

103:  IDENTIFIER IS NOT APPROPRIATE CLASS
104:  IDENTIFIER NOT DECLARED

End of compilation, number of errors=     2

FILE:  ERROR:P          HP Pascal/64000[A.1] Cross Ref Table


First occurence    Identifier        References
 6                  FACT              8,10,10
 2                  FACTORIAL
 4                  I                 4,9,10
 5                  N                 5
 9                  X                 9

Number of CASE's and BEGIN's  =    1
Number of END's               =    1

End of cross reference, number of symbols  =    5
```

The listing which follows is an example of an 8085 compiler listing to the printer with the expand option. Compiler listings differ slightly from this format depending on the setting of the compiler directive $LIST__OBJ$.

```
FILE:  SAMPLE:P        HP Pascal/64000[A.1] Expanded listing


  1  0000   1     "8085"
  2  0000   1     PROGRAM FACTORIAL;
                    0000          NAME "FACTORIAL Pascal"

  3  0000   1      VAR
  4  0000   1          I: INTEGER; {Loop control variable}
  5  0002   1          N: INTEGER;
  6  0004   1          FACT:  INTERGER;
  7  0006   1     BEGIN
                   0000       FACTORIAL:

  8  0000   1      FACT:= 1;
                   0000   31  ????    LXI   SP,STACK-
                   0003   21  0100    LXI   H,1
                   0006   22  ????    SHLD  FACTORIAL_D+4

  9  0009   1      FOR I:= 1 TO N DO
                   0009   2A  ????    LHLD  FACTORIAL_D+2
                   000C   22  ????    SHLD  FACTORIAL_D+6
                   000F   11  0100    LXI   D,1
                   0012   CD  ????    CALL  Zintleq
                   0015   CA  ????    JZ    FACTORIAL_L1
                   0018   EB          XCHG
                   0019           FACTORIAL_L2:

 10  001C   1      FACT:= FACT*I;
                   001C   EB          XCHG
                   001D   2A  ????    LHLD  FACTORIAL_D+4
                   0020   CD  ????    CALL  Zintmul
                   0023   22  ????    SHLD  FACTORIAL_D+4
                   0026   2A  ????    LHLD  FACTORIAL_D+6
                   0029   CD  ????    CALL  Zintneq
                   002C   CA  ????    JZ    FACTORIAL_L1
                   002F   EB          XCHG
                   0030   23          INX   H
                   0031   C3  ????    JMP   FACTORIAL_L2
                   0034           FACTORIAL_L1:

 11  0034   1     END.
                   0034   C3  ????    JMP   Z_END_PROGRAM
                   0037           FACTORIAL_C:
```

```
                   0037           FACTORIAL_E:
                   0037           FACTORIAL_D:
                   0037               DS     8
                   0037               GLB    FACTORIAL
                   003F               EXT    Zintmul
                   003F               EXT    Zintneq
                   003F               EXT    Zintleq
                   003F               EXT    STACK-
                   003F               EXT    Z_END_PROGRAM
                   003F               END    FACTORIAL

    End of compilation, number of errors=    0

     FILE: SAMPLE:P   HP Pascal/64000[A.1] Cross reference table

      First occurence       Identifier      References
       6                     FACT            8,10,10
       2                     FACTORIAL
       4                     I               4,9,10
       5                     N               5,9

      Number of CASE's and BEGIN's  =    1
      Number of END's               =    1

      End of cross reference, number of symbols =    4
```

NOTES

Chapter 10

LINKER INSTRUCTIONS

INTRODUCTION

A system application program, referred to as the linker (link), combines relocatable object modules into one file, producing an absolute image that is stored by the Model 64000 for execution in an emulation system or for programming PROMS. Interaction between the user and the linker remains basically the same regardless of which microprocessor assembler or compiler is being supported.

To prepare object code modules for the Model 64000 load program, the linker performs two functions:

a. Relocation: allocates memory space for each relocatable module of the program and relocates operand addresses to correspond to the relocated code.

b. Linking: symbolically links relocatable modules.

You may optionally select an output listing of the program load map and a cross-reference (xref) table. The linker also generates a listing that contains all errors that were noted. These error messages will contain a description of the error along with the file name and relocation/address information when applicable.

In addition to the above output listings, the linker constructs a global symbol file (link__sym type) and stores this file under the same file name assigned the absolute image/command file. This global file may be used for symbolic referencing during emulation. The link__sym file also contains the relocation address for all programs. This information is used to relocate asmb__sym type during emulation.

LINKER REQUIREMENTS

The following information is required by the linker:

a. File names of all object files to be loaded.

b. File names of libraries to be searched to resolve any unsatisfied externals.

c. Relocation information (load addresses for all relocatable areas).

d. Listing and debugging options as follows:

1) List (Load Map): file/program name, relocatable load addresses, and absolute load addresses.

2) Xref: symbols, value, relocation, and defining and referencing modules.

e. File name for command/absolute image file.

Since the linking operation will usually be required each time there is a software change and the information in items a through e remain constant for any given application, the linking control information is automatically saved in a command file with the same name as the absolute image file. The command file is distinguished from the absolute image file by file type.

LINKER SYNTAX

The command line in which Model 64000 commands are entered is accessed by way of the development station keyboard. Each system application function (edit, compile, assemble, link, emulate, etc.) can be called using keyboard softkeys. A syntax description of the link command follows.

link

SYNTAX

```
link [<FILE>] [listfile <list destination>]

     [options [edit][nolist][xref][no_overlap_check][comp_db]]
```

Default Values

<FILE>	If no linker command file is specified, the default allows creation of a new file of type link__com.
<list destination>	Defaults to user specified listfile default. See userid command.
options	If "options" is not entered, listing defaults to options specified in the linker command file.
	If options is specified, followed by nothing, a load map listing with no cross-reference is performed.

Examples:

link	Requests the linker to create a new linker command file. Listing output will go to the listfile default.
link PROGABS	Links absolute file PROGABS containing files in linker command file PROGABS. Listing output will go to listfile default and options in PROGABS type link__com are in effect.
link PROGABS options edit	Request the linker for purpose of viewing or modifying PROGABS:link__com. Listing output will go to listfile default.

link (Cont'd)

FUNCTION

The linker combines and relocates specified relocatable files creating an absolute file with the same name as that of the link__com file which can be used to program a PROM (with prom programmer option) or to load emulation RAM to be executed and analyzed with the emulator.

Parameters:

<FILE>	A file of type link__com to be used to direct the linker as to relocatable and relocation addresses.
<list destination>	File or device to which listing output is sent.
options	Allows user to override options specified in the linker command file.
nolist	Overrides the list option specified in the linker command file and suppresses output of a load map.
xref	Overrides no xref option specified in the linker command file and forces output of a global symbol cross-reference table.
edit	Allows user to edit existing link__com file specified.
no__overlap__check	Overrides overlap__check option specified in the linker command file and suppresses errors caused by memory overlaps. Default condition for linker overlap__check is ON.
comp__db	This file is created by the linker when requested and is a data base containing information from all of the comp__sym files associated with relocatables in an absolute file.

DESCRIPTION

The linker may be called by one of two methods: simple calling or interactive calling.

The simple calling method is used when interaction with an established command file is not required. That is, the current information in the command file is valid and no changes are required.

The interactive calling method is used when building a new linker command file or when the information in the current command file needs revision.

HOW TO USE THE LINKER

Simple Calling Method

a. Ensure that the following softkey prompts are displayed on the system CRT:

```
edit  compile  assemble  link  emulate  prom_prog  run  ---ETC---
```

b. Press the link softkey. The softkey configuration will be:

```
<CMDFILE>  listfile  options
```

c. The next prompt is **<CMDFILE>**. Type in the name of the established command file to be linked. The softkey configuration will change to:

```
           listfile  options
```

d. If it is necessary to change the output listing destination, press the listfile softkey. The softkey configuration will change to:

```
<FILE>  display  printer  null
```

e. Route the linker output listing to the desired location by selecting the <FILE> option, or by pressing the display softkey, the printer softkey, or the null softkey.

NOTE

Pressing the null softkey results in no output listing. Error messages will be displayed on the system CRT.

f. If the FILE option is desired in step e, type in the file name under which the listing is to be stored. You can then review your output listing on the system CRT using the edit function and your assigned file name.

g. The softkey configuration will change to:

```
                    options
```

h. Refer to the "options" default description in the LINK SYNTAX definition block.

i. If the [options] softkey is not used, the linker defaults to the list options specified in the command file and to noedit. To override the command file list options (for this link only), press the **[options]** softkey. The softkey configuration will change to:

```
[ edit ] [nolist] [ xref ] [no_overlap] [______] [______] [______] [______]
```

If only the [options] softkey is used, the linker defaults to list, noxref, and noedit. Any of these defaults may be changed by pressing the appropriate softkey.

j. After accomplishing step i, press the (RETURN) key. The linker will link the relocatable modules and produce the desired output listing.

Interactive Calling Method

The interactive calling method allows the user to create a new linker command file or edit an existing linker command file.

a. Ensure that the following softkey prompts are displayed on the system CRT:

```
[edit] [compile] [assemble] [link] [emulate] [prom_prog] [run] [---ETC---]
```

b. Press the [link] softkey. The softkey configuration will change to:

```
[<CMDFILE>] [listfile] [options] [______] [______] [______] [______] [______]
```

c. The user may start creating a new linker command file by not specifying any command file. An existing command file may be modified by specifying the command file name and the edit option.

NOTE

In the following paragraphes, the procedures are written for establishing a new command file. If an existing command file is being edited, just type in the changes required after each query. If no changes are required for a particular query, proceed to the next query. In all instances, to proceed to the next query, press the (RETURN) key.

d. The command query displayed in the command line of the system CRT is:

Object files? file1,file2,...,filen

The query asks for the names of the files to be linked and relocated. Type in the names of the files and then proceed to the next query.

NOTE

The softkey configuration "prompts" will change with each query from the linker. The softkey "prompts" indicate the type of information that is required.

Object files that are listed after the "Object files?" query may contain relocatable object modules, no-load files, and previously linked linker-symbol files (for global symbol references).

No-load files are differentiated from normal relocatable files by enclosing the no-load files in parentheses. Linker symbol files are specified by including the file type ':link__sym' in the file name.

Example:

```
FILE 1,(FILE 2,FILE 3),FILE 4:link__sym
```

NOTE

Refer to the paragraphs in this chapter that discuss no-load and link__sym files for additional information.

e. The next command query displayed in the command line on the system CRT is:

Library files? lib 1,lib 2,...,libn

Interrogation for library files is the same as for object files. After all object files have been linked, the linker determines if any external symbols remain undefined. The linker then searches the library files for object modules that define these symbols. The linker relocates and links only those relocatable modules that satisfy external references. Since a library file may contain more than one object module, all of its relocatable modules may not be linked. Refer to the paragraph in this chapter that discusses libraries and their construction.

NOTE

No-load files or linker symbol files, used for global referencing, must not be listed after this query. The no-load and link__sym files can only be referenced during the "Object files?" query.

After typing in the list of reference library files (or if library files are not referenced in the program), proceed to the next query.

f. The next command query displayed in the command line on the system CRT is:

Load addresses:PROG,DATA,COMN=addr,addr,addr

This query allows selection of separate, relocatable memory areas for the different modules of the program. For example, if you type in the following addresses:

Load addresses:PROG,DATA,COMN=1000H,2000H,3000H

the linker will relocate the PROG file module to memory location starting at address 1000H, the DATA module will be relocated to memory location starting at address 2000H, and the COMN module will be relocated to memory location starting at address 3000H.

NOTE

Load addresses may be entered using any number base (binary, octal, decimal, or hexadecimal); however, the addresses listed in the load map are give in hexadecimal only.

The default addresses are zeros. After entering the load addresses or if the default addresses are acceptable, proceed to the next query.

g. The next command query displayed in the command line on the system CRT is:

More files? no

The linker asks if more files are to be linked. If the response is yes, the linker begins interrogation again, allowing additional object and library files to be specified with new load addresses. When specifying new relocatable areas, the user may continue with the previously relocatable area by typing "CONT" in the appropriate field (or using the CONT softkey). The relocatable area is treated as if no new address was assigned.

Example:

Load addresses:PROG,DATA,COMN=0BCCH,CONT,3FFCH

The default condition to the "more files?" query is no. Proceed to the next query.

h. The next command query displayed in the command line on the system CRT concerns output listing options. It has the following syntax:

List,xref,overlap__check=on off on

The linker asks you to specify what output listings are required and if memory overlap should be checked. Using the on or off softkey select in the sequence indicated in the syntax statement (list, xref, overlap__check), the desired output listing and memory check condition. After inserting the requirements, proceed to the next query.

NOTE

The output listings indicated after the list,xref,overlap__check= query are the command file values that will be used during this and future link operations. They can be overridden by using the options softkey during the linker call.

The default condition for this query is **on off on**.

i. The next command query displayed in the command line on the system CRT is:

Absolute file name=name

This final query from the linker allows you to assign a name to the new command/absolute image file that you are about to link. The absolute image file that is created by the linker is always associated with a link command file of the same name. A global symbol file is also established under the name of the command/absolute image file name. The global symbol file contains all global symbols and their relocation values.

After entering the absolute file name, press the (RETURN) key.

The linker will link, relocate the files, and save the linking information in the command file.

LINKER OUTPUT

The linker listings may be output to the system display, line printer, or any file. The following information may be included in the linker output listing:

a. List (Load Map)

b. Cross-reference table

c. Error messages

NOTE

Certain error messages contain more than 80 characters and will not be completely displayed on the system CRT. However, complete error messages will be printed when using the line printer or a list file for listings.

List (Load Map)

A load map is a listing of the memory areas allocated to each relocatable file. The listing begins with the first file linked and proceeds to list all other linked files with their allocated memory locations. An example of a load map listing that will be printed on the system printer is as follows:

```
FILE/PROG NAME   PROGRAM  DATA  COMMON  ABSOLUTE   DATE                TIME COMMENTS
-------------------------------------------------------------------------------------

KYBD:SAVE         0000                             Thu,5 Jun 1980, 11:37
EXCT:SAVE                               0B00-0B34 Thu,5 Jun 1980, 10:38
DSPL:SAVE                 A100                     Thu,5 Jun 1980, 11:38

next address      0021    A121

REG1:SAVE         B000                             Thu,5 Jun 1980, 11:52
REG2:SAVE         B103                             Thu,5 Jun 1980, 11:53
REG3:SAVE         B206                             Thu,5 Jun 1980, 11:58

next address      B30C


Libraries
PARAMETER:SAVE    0021                             Thu,5 Jun 1980, 11:43
MULTEQUAT:SAVE    0221                             Thu,5 Jun 1980, 11:45

next address      0421    A121

XFER address=0B00 Defined by EXCT
No. of passes through libraries= 1
absolute & link_com file name=SETAG1:SAVE
Total# of bytes loaded= 0782
```

A brief description of each column in the listing is as follows:

a. FILE/PROG NAME – this column will contain the name of the files that are linked. In the event library files are referenced, not only will the master library file be listed, but its subsections will be indented to indicate that they are part of the main library file. No-load files will be displayed in parentheses (...).

b. PROGRAM - this column will indicate the first address (hexadecimal) of a memory block that contains the PROG relocatable code in the file listed in the FILE/PROG NAME column.

c. DATA - this column will indicate the first address (hexadecimal) of a memory block that conatins the DATA relocatable code in the file listed in the FILE/PROG NAME column.

d. COMMON - this column will indicate the first address (hexadecimal) of a memory block that contains the COMN relocatable code in the file listed in the FILE/PROG NAME column.

e. ABSOLUTE - this column will indicate the hexadecimal addresses of a memory block that contains the absolute code assigned by the file listed in the FILE/PROG NAME column.

NOTE

The "next address" statement in the load map listing indicates the next available hexadecimal address in the PROG, DATA, or COMN memory areas. It may also be used to determine the number or bytes (words for 16-bit processors) that are contained in each area (next address - starting address = total bytes).

f. DATE - this column will indicate the date that the file listed in the FILE/PROG NAME column was assembled (assuming the system date/time clock was current).

g. TIME - this column will indicate the time that the file listed in the FILE/PROG NAME column was assembled (assuming the system date/time clock was current).

h. COMMENTS - this column will contain user comments entered during assembly by the assembler pseudo NAME instruction.

Cross-reference Table

The cross-reference table lists all global symbols, the relocatable object modules that define them, and the relocatable modules that reference them. An example of a cross-reference listing that will be listed on the system printer is as follows:

```
SYMBOL     R     VALUE     DEF BY        REFERENCES

DSPL6      P     0034      PGM68D        PGM68E
KYBD6      P     0001      PGM68K        PGM68E
```

A brief description of each column in the cross-reference listing is as follows:

a. SYMBOL - all global symbols will be listed in this column.

b. R (Relocation) - in this column a letter will identify the type of program module. The letters that are available and their definitions are:

A = Absolute
C = Common (COMN)
D = Data (DATA)
P = Program (PROG)
U = Undefined

c. VALUE - relocated address of the symbol.

d. DEF BY - this column will contain the file name that defines the global symbol.

e. REFERENCES - this column will list the file names that reference the global symbol.

"NO-LOAD" FILES

Files that are enclosed in parentheses in the "Object files?" query indicates to the linker that no code is to be generated for the file. Relocation and linking occurs in the same manner as if the file was a load file; however, the absolute image file generated by the linker does not contain the object code for the no-load file. No-load files may be useful in linking to existing ROM code or in the design of software systems requiring memory overlays.

LINKER SYMBOL FILE

The linker creates a global symbol file for every link operation. The global file name is the same as the assigned command/ absolute image file name assigned to the link. The user may find that linking to a common piece of code (global) is simplified by referring to that code by its linker-symbol file. This is accomplished by referencing the correct linker-symbol file name during the "Object files?" query by the linker. The linker-symbol file name referenced at the time of the query must be specified by the type ':link__sym'.

```
Object files? PGM68k,Pgm68D:link__sym
```

LIBRARY FILES

Libraries are a collection of relocatable modules that are stored on the system disc and may be referenced by the linker.

If a library file name is given as a response to the "Object files?" query, all the relocatable modules in the library file will be relocated and linked. If a library file name is given as a response to the "library files?" query, only those relocatable modules that define the unsatisfied externals will be relocated and linked. The remaining relocatable modules in the library file are ignored.

It is possible to combine relocatables into a library by using the system library command. Refer to the System Software Reference Manual for a detailed description of the library command.

ERROR MESSAGES

When an error is detected during the link process, the linker will determine if the error is fatal or nonfatal. If the error is classified as fatal, the linker will abort the linking process. If the error is nonfatal, the linker will continue the linking process, but will generate error messages that will be listed in the output listing. A description of each error message is given in the following paragraphs.

Fatal Error Messages

Upon encountering a fatal error the linker will display one of the following messages on the system CRT STATUS line. The linker will abort the link process and return control of the system to the monitor.

a. **Out of Memory in Pass 1.**

The linker will issue this message to indicate that there is insufficient memory to accommodate the current operation. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

NOTE

As a general rule, the available memory space can handle programs containing apporoximately 3000 symbols. However, if cross-reference symbol tables are required, the symbol handling capability is reduced to approximately 1500 symbols.

b. **Out of Memory in Pass 2.**

The linker will issue this message to indicate that there is insufficient memory to accomodate the current operation. To correct this siruation, reduce the number of files, global symbols, and/or external symbols used during the current link.

c. **Out of Memory in Xref.**

The linker will issue this message to indicate that there is insufficient memory to accommodate the building of a cross-reference table. This error does not affect the absolute file since it is created and stored prior to the linker attempting to build the cross-reference file. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

d. **Target Processors Disagree.**

The linker will issue this message if the relocatable modules to be linked are designed for different processors. Ensure that all relocatable modules assigned for linking are written for the same type microprocessor.

e. **Checksum Error.**

The linker will issue this message if it is unable to read a relocatable file due to a checksum error or other irregularities in the file. To correct this situration, reassemble the relocatable file, then, re-link.

f. **Linker System Error.**

The linker will issue this message if it detects a hardware or software failure in the Model 64000. To correct this situation re-link the relocatable modules or run the hardware performance verification program.

g. **File Manager Errors.**

The linker will issue certain messages if the system file manager is unable to perform the specified file operation as requested by the linker. Refer to the System Software Reference Manual (Appendix A) for a list of File Manager Errors.

Nonfatal Error Messages

Upon encountering nonfatal errors, the linker will continue the link operation and print the error messages (except initialization errors) in the output listing. An error message that is listed will contain a description of the error and the name of the file where the error occurred. If the null list file is in effect, the linker will direct the error messages to the data area of the system CRT.

a. **Illegal entry: re-enter.**

During initialization the linker will indicate in the STATUS line on the system CRT that the user has made an illegal response to an interrogation. To correct this situation, re-enter the proper response.

b. **Duplicate symbol.**

During pass 1 of the link process, the linker detects that the same symbol has been declared global by more than one relocatable module. The first definition holds true. The relocatable module that first defines the symbol may be found in the cross-reference table. To correct this error, remove the extra global declarations.

c. **Load address out of range.**

The linker has tried to relocate code beyond the addressing range of the specified microprocessor. To correct this situation, reassign the relocatable addresses.

d. **Multiple transfer address.**

During pass 1, the linker finds that the transfer address has been defined by more than one relocatable module. The first definition holds true. The relocatable module that first defined the transfer address will be given at the conclusion of the linking. To correct this situation, remove the extra transfer address. Reassemble the amended relocatable module; then, re-link. If a xfer address is defined by both a no-load program and a load program, no error will be given. The load program xfer address takes precedence.

e. **Undefined symbol.**

During pass 2, the linker finds that a symbol has been declared external but not defined by a global definition. To correct this situation, define the symbol.

f. **Out of memory in xref.**

Unlike the fatal error (Out of Memory in Xref), this error occurs when memory space is available for a complete symbol table but only a portion of the cross-reference table. The linker will complete the xref operation, listing only that portion of the cross-reference table for which memory space was available. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

g. **Memory overlap.**

Relocatable program areas have been overlapped in memory. The error message will list the program names and the overlapping areas.

h. **Address out of range.**

The operand address is not within a valid addressing range for the specific microprocessor involved.

NOTES

Appendix A

Compile Time Errors

The following errors are detected by the first pass of the compiler. Errors are also detected by the second pass of the compiler. These errors are microprocessor dependent and are listed in the microprocessor-dependent supplements.

When errors appear in groups, usually only the first message is meaningful. This is so because some of the following error messages appear as a result of the first error. In particular, any time the WARNING message (number 0) is indicated, the compiler will attempt to resume compilation at the next logical token. In some instances, correctly. In these situations, the user should use the editor function to correct the first error.

LIST OF ERROR MESSAGES

0: **WARNING**: attempted syntax error recovery here
1: Error in simple type
2: Identifier expected
3: **PROGRAM** expected
4: ')' expected
5: ':=' expected
6: Illegal symbol
7: Error in parameter list
8: **OF** expected
9: '(' expected
10: Error in type
11: '[' expected
12: ']' expected
13: **END** expected
14: ';' expected
15: Integer expected
16: '=' expected
17: **BEGIN** expected
18: Error in declaration part
19: Error in field list
20: '.' expected
21: '*' expected
22: **LABEL** expected
23: **CONST**, **TYPE**, **VAR**, **BEGIN**, **FUNCTION**, or **PROCEDURE** expected
24: **EOF** expected
25: Statement BEGIN symbol expected

LIST OF ERROR MESSAGES (Cont'd)

26: **PROCEDURE** or **FUNCTION** expected
27: ';' or OTHERWISE expected
28: '(' or '[' expected
29: String expected
30: Type name expected
31: '..' expected
32: Error in variant label

50: Error in constant
51: ':' expected
52: **THEN** expected
53: **UNTIL** expected
54: **DO** expected
55: **TO/DOWNTO** expected
56: **IF** expected
57: **FILE** expected
58: Error in factor
59: Error in variable

101: Identifier declared twice
102: Low bound exceeds high bound
103: Identifier is not of appropriate class
104: Identifier not declared
105: Sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
109: Type must not be **REAL**

110: Tag field type must be scalar or subrange
111: Incompatible with tag field type
112: Index type must not be **REAL**
113: Index type must be scalar or subrange
114: Base type must not be **REAL**
115: Base type must be scalar or subrange
116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Forward declared: repetition of parameter list not allowed

LIST OF ERROR MESSAGES (Cont'd)

120: Function result type must be scalar, subrange, or pointer
121: File value parameter not allowed
122: Forward declared function; repetition of result type not allowed
123: Missing result type in function declaration
124: F-format for **REAL** only
125: Error in type of standard function parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type of parameter function does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types not compatible
138: Type of variable is not array
139: Index type is not compatible with declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter substitution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must not be integer
150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable must not be declared on intermediate level
156: Multidefined case label
157: Too many cases in case statement
158: Missing corresponding variant declaration
159: Real or string tag fields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed

LIST OF ERROR MESSAGES (Cont'd)

165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was redeclared
172: Undeclared external file
175: Missing file **INPUT** in program heading
176: Missing file **OUTPUT** in program heading
177: Assignment to function identifier not allowed here
178: Multidefined record variant
180: Control variable must not be formal
182: For parameter of form E1:E2:E3, E2 and E3 must be INTEGER compatible
183: Parameter of form E1:E2 or E1:E2:E3 not allowed here
184: Parameter of form E1:E2:E3 not allowed here
185: Illegal to dereference function result
186: Illegal to select element of function result
188: Variant may not contain files

201: Error in real constant: digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number
205: Zero string not allowed
206: Integer part of real constant exceeds range

250: Too many nested scopes of identifiers
251: Too many nested procedures and/or functions
252: Too many forward references or procedure entries
254: Too many long constants in this procedure
255: Too many errors on this source line
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated
260: Too many exit labels

270: # not in column 1 or **PREPROCESS** not specified. Remainder of line ignored

280: Preprocessor syntax error
281: Unimplemented preprocessor instruction
282: Number of parameters does not agree with macro declaration
283: Identifier not **#DEFINED**
284: Macro may not have more than 20 parameters

LIST OF ERROR MESSAGES (Cont'd)

285: **#IF** without **#ENDIF**
286: #IF instruction may not contain multiline macro
287: **#ELSE** or **#ENDIF** without **#IF**
288: Preprocessor stack overflow. Simplify constant expression
289: Error in constant expression

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range
305: Implementation restricts case constants to 16 bit values
398: Implementation restriction
399: Variable dimension arrays not implemented

400: State stack overflow; break program into modules
401: Previous error has resulted in unrecoverable parser error
402: End of source before end of compilation
403: Symbol table overflow; beware of scalar types with many scalars
404: Semantic stack overflow; break program into modules
405: End of source before end of comment
406: Out of expression tree storage; simplify expression
407: Pop of empty semantic stack; probably caused by previous error
408: Illegal entry on semantic stack; caused by previous error
409: Label may not have more than four digits

410: Too many indirect; simplify expression
411: Constant expression expected
412: More than 20 syntax errors; parse aborted
413: Assignment to constant
414: More than 255 subroutines; break program into modules
415: Files not implemented
416: More than 255 large constants

450: Feature not implemented
451: Structured constants not implemented
452: Sets bigger than 16 elements not implemented
453: Sets subranges not implemented
454: Label subranges not implemented
455: Language extensions used in extensions off mode
456: Too many user defined operation types
457: This option allowed in program level type section only

LIST OF ERROR MESSAGES (Cont'd)

500: **WARNING:** illegal compiler option; option ignored
501: **WARNING:** packing not implemented
502: **WARNING:** smallest set element < 0; 0 assumed
(8080/8085 amd Z80 compilers only)
503: **WARNING:** largest set element > 15; 15 assumed
(8080/8085 and Z80 compilers only)
504: **WARNING:** non-standard feature used
505: **WARNING:** type change changes physical size
506: **WARNING:** +32768 is treated as -32768 by the compiler
(8080/8085 and Z80 compilers only)
509: **WARNING:** compiled as 0...(upper bound)
512: **WARNING:** expanded line larger than 240 characters; multiple
lines created
513: **WARNING:** duplicate macro name; New definition holds

Index

The following index lists important terms and concepts of this manual along with the location(s) in which they can be found. The numbers to the right of the listings indicate the following manual areas:

- Chapters – references to chapters appear as "Chapter X", where "X" represents the chapter number.
- Appendices – references to appendices appear as "Appendix Y", where "Y" represents the letter designator of the appendix.
- Figures – references to figures are represented by the capital letter "F" followed by the section figure number.
- Other entries in the Index – references to other entries in the index are preceded by the word "See" followed by the reference entry.

p (Cont'd)

64800-90909, JANUARY 1984
Replaces: 64800-90908, June 1982

HEWLETT
PACKARD

UPDATED: OCTOBER 1985
PRINTED IN U.S.A.